Revisionist Integral Deferred Correction Methods with Application to the Moving Method of Lines

by

© **Bilal Uddin**

A thesis submitted to the School of Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science.

Department of Mathematics and Statistics

Memorial University

September 2018

St. John's, Newfoundland and Labrador, Canada

Abstract

Revisionist integral deferred correction (RIDC) methods are time parallel predictor-corrector methods used to solve initial value problems (IVPs). The prediction and the correction formulae are designed in such a way so that the prediction and the correction steps can be computed simultaneously. More than one computing core can be used at a time to correct the approximate solutions at different correction levels. The multi-core implementation can improve the efficiency of the methods in terms of runtime.

In our thesis, we ultimately wish to solve parabolic partial differential equations (PDEs) numerically by combining the spatial adaptive moving mesh method with the time parallel revisionist integral deferred correction (RIDC) method. To do so, we expand an existing RIDC library to handle systems of IVPs of the form $L(t,y)y^{'} = f(t,y),\ y \in \mathbb{R}^n$. Discretization of a physical PDE by the moving method of lines coupled with a semi-discretized moving mesh PDE results in a system of IVPs of the form $L(t,y)y^{'} = f(t,y)$, where $y(t)$ is a vector consisting of the physical solution $u \in \mathbb{R}^n$ and the mesh $x \in \mathbb{R}^n$, and $L(t,y)$ is a state dependent square matrix. We achieve a RIDC implementation for this family of IVPs by systematically expanding the existing RIDC formulation and software. We have verified our derived formulae and software with relevant examples.

Dedicated to my loving parents and my siblings

Lay summary

Many mathematical models of physical systems are written as partial differential equations (PDEs). Analytic solutions of many partial differential equations (PDEs) are either not available or are computationally very complicated. For this reason, numerical approximations to the solution of PDEs are of great importance in applied and computational mathematics. The numerical solution of a PDE is computed at some discrete points known as mesh points. If the mesh points are fixed and uniformly spaced then the mesh is called uniform mesh. Solutions of many physical PDEs using uniform mesh have rapid variations in the solutions. In order to mitigate those solution variations, mathematical formulae are designed so that the mesh points can move in response to the changes in physical solution. That is mesh points automatically move towards the regions with high solution variations. A mathematical formulation of producing such movable mesh is called moving mesh method.

The efficiency of a numerical method greatly depends on how long it takes to compute the solution. The computational time can be reduced if several steps or parts of the method can be computed in parallel on a parallel computer. The revisionist integral deferred correction (RIDC) method is such a time parallel method for the numerical solution of initial value problems (IVPs).

In our thesis, we discretize a PDE by finite difference method on moving mesh and obtain a coupled system of IVPs. We then solve the resulting system of IVPs by the time parallel RIDC method and extend the existing RIDC library to system of IVPs of that form.

Acknowledgements

I would like to thank to my supervisor, Dr. Ronald Haynes, for his guidance, motivation and advice throughout the project. I would also like to thank to Dr. Benjamin Ong, Michigan Technological University for his valuable time and contribution to this research work. Special thanks to Dr. Weizhang Huang for his valuable suggestions on moving mesh method.

I am very grateful to all the faculty and staff members of the Department of Mathematics & Statistics at Memorial University for their support during my last 2 years study.

Finally, I am thankful to my parents, siblings and friends for their constant support and inspiration during my study period at Memorial University.

Statement of contribution

I would like to acknowledge that

- All the contents of the thesis including the writing, computer codes typed in Matlab and C++ are joint work with my supervisor Dr. Ronald Haynes.
- The original RIDC library we have used as the base for our code is owned by Dr. Benjamin Ong and Dr. Ronald Haynes.
- All other books and articles, which we have used, are cited at the respective places in the thesis.

Table of contents

List of tables

List of figures

Chapter 1

Introduction

This chapter provides an overview of different families of numerical methods for solving ordinary differential equations (ODEs) and partial differential equations (PDEs). At the beginning of the chapter, we focus on the well-known sequential time stepping methods used to solve ODEs followed by a brief discussion on parallelism and adaptivity used to improve the efficiency and accuracy of numerical methods. The last part of the chapter provides background on the integral deferred correction methods and the revisionist integral deferred correction (RIDC) methods which are the focus of this thesis. The chapter concludes by giving the objectives and an outline of the remainder of the thesis.

We start with the following initial value problem

$$y'(t) = f(t, y(t)),\ t \in [t_0, T],\ y(t_0) = y_0, \tag{1.1}$$

where $y \in \mathbb{R}^n$ and f is a vector-valued function $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$. We first discuss some of the well known sequential time stepping methods used to solve (1.1) numerically. The discussion is based on the books by Ascher and Petzold [1], and by Randall J. LeVeque [34].

Let $y_0, y_1, \ldots, y_N$ be the approximate solutions of the initial value problem (1.1) at time nodes $t = t_0, t_1, \ldots, t_N = T$ respectively, with a given initial value y_0, where N is the total number of the time intervals.

The sequential time stepping methods for the numerical solution of (1.1) are classified as

(i) One-step methods: In a one-step method, y_{n+1} is approximated using only one previously computed value y_n , for $n = 0, 1, \ldots, N-1$. Euler's method, Taylor series methods and Runge-Kutta methods are examples of one step methods.

(ii) Linear multi-step methods: In a *r-step* linear multi-step method, y_{n+r} is approximated using one or more previously computed values $y_{n+r-1}, y_{n+r-2}, \ldots, y_n$, for $n = 0, 1, \ldots, N-1$. Adams methods and Backward Differentiation Formulas (BDF) are examples of linear multi-step methods.

The most elementary one-step method for the solution of the initial value problem (1.1) is Euler's Method. To construct Euler's method, we discretize the given time domain $[t_0, T]$ into N equally spaced intervals. We assume that $t_0, t_1, \ldots, t_N$ are the time nodes and $h = \frac{T-t_0}{N}$ is the uniform stepsize, and the function f is sufficiently smooth. Using Taylor's expansion of $y(t)$ about $t = t_n$, we have

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \mathcal{O}(h^3), \tag{1.2}$$

where $\mathcal{O}(h^3)$ means that in the limit $h \to 0$, the dominant term is of the form ch^3 for some constant c. Taking the first two terms of (1.2) and ignoring the remaining terms, we have the approximation

$$y(t_{n+1}) \approx y(t_n) + hy'(t_n). \tag{1.3}$$

Replacing y' by f, we obtain the explicit Euler or Forward Euler method given as

$$y_{n+1} = y_n + hf(t_n, y_n),\ n = 0, 1, \ldots, N-1. \tag{1.4}$$

Similarly, from the Taylor's expansion

$$y(t_n) = y(t_{n+1}) - hy'(t_{n+1}) + \frac{h^2}{2}y''(t_{n+1}) + \mathcal{O}(h^3), \tag{1.5}$$

we obtain the implicit Euler or backward Euler method given by

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}),\ n = 0, 1, \ldots, N-1. \tag{1.6}$$

From equations (1.4) and (1.6) we notice that a forward Euler method requires only a single function evaluation whereas backward Euler method requires the solution of

a non-linear system of equations (if f is non-linear) at each time step. The backward Euler method is computationally more expensive per step when it is implemented for non-linear problems. However, the backward Euler method can give significant advantages over forward Euler method in terms of stability and stepsize selection for the same problems.

Euler's method has a very simple structure and per-step it is quite inexpensive. However, Euler's method is only first order accurate and this limitation leads us to find a method that can provide greater accuracy for the same step size.

Taylor series methods are a family of one-step methods which can provide high order solutions to initial value problems. An arbitrary order Taylor series method can be derived (shown in [1]) from the following Taylor series approximation

$$y(t_{n+1}) \approx y(t_n) + hy'(t_n) + \frac{h^2}{2}y''(t_n) + \cdots + \frac{h^p}{2}y^{(p)}(t_n). \tag{1.7}$$

In the approximation (1.7), the Taylor series remainder term is neglected. From equation (1.1), we have

$$y'(t_n) = f(t_n, y(t_n)). \tag{1.8}$$

Differentiating (1.1), we have

$$y''(t_n) = [f_t + f_y y']_{(t_n,y_n)} = [f_t + f_y f]_{(t_n,y_n)}, \tag{1.9}$$

where the subscript (t_n, y_n) indicates that the expression to the left is evaluated at $t = t_n$ and $y = y_n$. Continuing this process, we have

$$y'''(t_n) = [f_{tt} + 2f_{ty}f + f_t f_y + f_{yy}f^2 + f_y^2 f]_{(t_n,y_n)}, \tag{1.10}$$

and so on.

Substituting the derivatives $y'(t_n), y''(t_n), \ldots, y^{(p)}(t_n)$ in equation (1.7) we can obtain the Taylor series method of order p.

One of the major problems with Taylor series methods is that we need to evaluate higher order derivatives of f, and finding the derivatives of f is difficult for many practical problems [1]. This leads us to find high order, derivative free, one-step

methods called Runge-Kutta methods. An *r-stage* Runge-Kutta method is given by

$$\begin{aligned} k_i &= f(t_n + c_i h,\, y_n + h\sum_{j=1}^{r} a_{ij}k_j),\ i = 1, 2, \ldots, r, \\ y_{n+1} &= y_n + \sum_{i=1}^{r} b_i k_i. \end{aligned} \tag{1.11}$$

The method (1.11) is determined by its coefficients, which are collected in a Butcher tableau [1] as

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \ldots & a_{1r} \\ c_2 & a_{21} & a_{22} & \ldots & a_{2r} \\ \vdots & \vdots & & & \vdots \\ c_r & a_{r1} & a_{r2} & \ldots & a_{rr} \\ \hline & b_1 & b_2 & \ldots & b_r \end{array}, \text{ where } c_i = \sum_{j=1}^{r} a_{ij},\ i = 1, 2, \ldots, r.$$

The Runge-Kutta method (1.11) is explicit if $a_{ij} = 0$, whenever $j \geq i$, otherwise it is implicit. One of the drawbacks of Runge-Kutta methods is that an *r-stage* Runge-Kutta method requires r function evaluations per step. This may create difficulties if function values are difficult or expensive to evaluate. High order Runge-Kutta methods can also be expensive particularly for the implicit case [34]. In general, calculation of k_i in (1.11) is a sequential process.

These drawbacks of Runge-Kutta methods motivate us to use a multistep method where only one function evaluation is required per time step. Previously computed function values are used to obtain higher order accuracy.

An *m-step* linear multistep method [1] applied to the initial value problem (1.1) is given

$$\sum_{j=0}^{m} \alpha_j y_{n+j} = h\sum_{j=0}^{m} \beta_j f_{n+j}, \tag{1.12}$$

where α_j, β_j are the method coefficients, $f_j = f(t_j, y_j)$, $t_j = t_0 + jh$, $h = \frac{T-t_0}{N}$ and $\alpha_m \neq 0$. For an *m-step* linear multi-step method, we require m starting values $y_j = y(t_j)$, $j = 0, 1, \ldots, m-1$. In equation (1.12), if $\beta_m = 0$, then the method is explicit, otherwise it is implicit. The value y_{n+m} is computed from this equation in terms of the previous values $y_{n+m-1}, y_{n+m-2}, \ldots, y_n$.

If $\alpha_m = 1$, $\alpha_{m-1} = -1$, and $\alpha_j = 0$ for $j < m-1$, we get the Adams methods (given in [34]) of the form

$$y_{n+m} = y_{n+m-1} + h \sum_{j=0}^{m} \beta_j f_{n+j}. \tag{1.13}$$

If $\beta_m = 0$, the formula (1.13) is called explicit Adams method or Adams-Bashforth (AB) method. An *m-step* AB method can alternatively be derived by interpolating f through the m points $t = t_i, t_{i-1}, \ldots, t_{i+1-m}$ [34]. Since m interpolation points are used, an *m-step* AB method is expected to be m^{th} order accurate.

If $\beta_m \neq 0$, formula (1.13) is called implicit Adams method or Adams-Moulton (AM) method. An *m-step* AM can alternatively be derived by interpolating f through the $m+1$ points $t = t_{i+1}, t_i, t_{i-1}, \ldots, t_{i+1-m}$ [34]. Since $m+1$ interpolation points are used, an *m-step AM* method is expected to be $m+1$ order accurate.

Another family of linear multistep method is the Backward Differentiation Formulas (BDF). For an *m-step* BDF method we evaluate f at the right end of the current step, (t_{i+1}, y_{i+1}), and construct an interpolating polynomial of y passing through the points $t = t_{i+1}, t_i, t_{i-1}, \ldots, t_{i+1-m}$, and finally differentiate the interpolation polynomial to obtain the BDF formula. A BDF method derived using m interpolation points gives an implicit method of order m [1].

All of the methods mentioned above are typically implemented in a sequential way. For example, y_{n-1} is needed before y_n is computed. The function values f_n or stage values are also generally computed sequentially.

We now describe the commonly used parallel methods for solving ODEs. Burrage [4, 5] classifies parallel methods for the numerical solution of ODEs into the following three categories:

(a) *Parallelism across the system*: The original system is subdivided into a number of smaller systems so that the resulting subsystems can be solved in parallel. Waveform relaxation methods [42, 43] are examples of this parallelism technique.

(b) *Parallelism across the method*: Parallelism across the method allows concurrent function evaluations on different processors. This kinds of parallalism approach can be found in predictor–corrector based RIDC methods [9,13], and Runge-Kutta methods [30].

(c) *Parallelism across the step*: In this approach, the time domain is divided into subdomains and the equations are solved in parallel over steps. The parareal method [23] is an example of this approach.

A survey of parallel numerical methods for finding the roots of nonlinear equations, solving the differential equations, and solving systems of linear equations was carried out by Miranker [37]. Another survey on parallel numerical methods, specifically for IVPs, was performed by Jackson [29]. In 2015, Gander [20] wrote an article on time parallel methods for numerical time integration and divided the time parallel methods into the following four categories:

(i) *Domain decomposition methods in space-time*: In this type of time-parallel methods, the given space and time domains are decomposed into subdomains. Solutions in each subdomain (in space) are computed in parallel (over time). Overlapping and non-overlapping Schwarz waveform relaxation methods [19, 21] are examples of this class of time parallel methods.

(ii) *Shooting type time parallel methods*: An example of this class is the *multiple shooting method*, in which time interval $[t_0, t_f]$ is divided into n subintervals and shooting technique is carried out on each subinterval $[t_{i-1}, t_i]$, $i = 1, \ldots, n$. The shooting method applied to each subinterval results in a nonlinear system of equations, $F(u) = 0$, which can be solved by Newton's method. At each iteration of Newton's method, the evaluation of the function value $F(u)$ and its Jacobian $F^{'}(u)$ can be computed in parallel. This type of parallelism is studied in [3, 6, 38].

(iii) *Multigrid methods in space-time*: Parallel space-time multigrid methods for parabolic problems are introduced by Gander and Neumuller [22] and Horton and Vandewalle [25]. Further space-time parallel multigrid methods can be found in [15, 20]. Multigrid reduction in time algorithm (MGRIT) [17, 18] is another parallel-in-time multigrid method.

(iv) *Direct solvers in space-time*: Time parallel predictor-corrector methods fall in this category, where prediction and correction steps can be computed concurrently. RIDC methods developed by Christlieb, Macdonald and Ong [13] are examples of this direct time parallel approach.

RIDC methods fall in category (iv), and will be the focus of the thesis. In order to introduce the RIDC methods, we start with the spectral deferred correction (SDC) methods for the numerical solution of ordinary differential equations (ODEs) introduced by Dutt, Greengard and Rokhlin [14].

The ODE (1.1) is converted into the equivalent Picard integral equation

$$y(t) = y_0 + \int_{t_0}^{t} f(\tau, y(\tau))d\tau. \tag{1.14}$$

The residual function $r(t)$ is obtained by substituting an approximate solution, $u(t)$, in the integral equation (1.14)

$$r(t) = y_0 + \int_{t_0}^{t} f(\tau, u(\tau))d\tau - u(t). \tag{1.15}$$

Combining (1.14) and (1.15) and some algebraic calculation allows us to show that the error $e(t) = y(t) - u(t)$ satisfies the following integral form

$$e(t) = r(t) + \int_{t_0}^{t} \Big(f(\tau, u(\tau) + r(\tau)) - f(\tau, u(\tau)\Big)d\tau. \tag{1.16}$$

The Picard integral equation (1.14) and the error equation (1.16) may be approximated by the Euler method at $m + 1$ nodes: $t_0 < t_1 \dots t_m < t_{m+1} = T$. At each time point the error equation is solved and the approximate solution is corrected by $u^{[l]} = u^{[l-1]} + e^{[l]}$, where l is the number of corrections requested by the specific method. To approximate the definite integral in the error equation, Gaussian quadrature nodes in the interval $[-1, 1]$ are used. It was shown that each correction of the SDC method using a first order integrator improves the order of accuracy of the solution by one order. Further studies on SDC methods, including convergence and stability of the methods, can be found in [2, 24, 32, 35, 36]. Semi-implicit SDC methods for solving ODEs were developed by Minion et al. in [36]. The choice of quadrature nodes and the choice of predictors for the semi-implicit SDC methods were studied in [33] and [31].

Christlieb, Ong and Qiu, in [10, 11], experimented with spectral deferred correction (SDC) methods with high order integrators and various choices of quadrature points. In those papers they re-constructed SDC methods using high order integrators and equally spaced quadrature points, and named them integral deferred correction

(IDC) methods. In fact when IDC methods are constructed using non-uniform Gaussian quadrature the methods coincide with SDC methods. In [11], IDC methods were formulated using high order Runge-Kutta (RK) integrators and equally spaced quadrature points instead of Gaussian quadrature points, and [10] formulated IDC methods with several high order integrators including multi-step methods. A comparative study of IDC methods and Runge-Kutta (RK) methods was also given in [10]. Semi-implicit IDC methods were developed in [8].

Later in 2010, Christlieb, Macdonald and Ong [13] showed that they were able to compute the prediction and correction steps of IDC methods in parallel and they named the resulting parallel-in-time method a revisionist integral deferred correction (RIDC) method. According to [13], the formulation of RIDC method is similar to that of the IDC method except for the variation in the choice of number of subintervals in each group and the way the computation is completed in the correction loop.

The time interval $[0, T]$ is divided into N equally spaced intervals, and the resulting N intervals are further partitioned into J groups of intervals I_j, $j = 1, \ldots, J$, so that each group I_j contains $K(K >> M)$ subintervals, where $M(= p - 1)$ is the number of corrections required by a p^{th}-order RIDC method. For a p^{th}-order RIDC method the number of subintervals K in each group can be much larger than M, whereas, in IDC methods K is always equal to M. The time loop in each group I_j, $j = 1, \ldots, J$, is split into two separate loops. One loop runs from $m = 1, \ldots, M$, and another loop runs from $m = M + 1, \ldots, K$. The two individual loops enable the correction loop to be executed in parallel. Like other deferred correction methods, RIDC methods work sequentially over group of intervals, I_j, $j = 1, \ldots, J$, starting with the initial group of intervals, I_1. The solution obtained at the end of the group I_1 is used as the initial solution for the group I_2, the solution obtained at the end of the group I_2 is used as the initial solution for the group I_3, and we proceed this way until we reach to the last group I_J. The available choice of integrator and quadrature formulae for RIDC methods are similar to those available for IDC methods. See Chapter 2 for more details.

For the solution of stiff problems explicit RIDC methods are no longer good numerical schemes unless the time step is very small. Implicit RIDC methods, that is RIDC methods constructed using backward Euler (for example) as the predictor and corrector, were further developed by Christlieb and Ong in [9]. This method

can handle both stiff and non-stiff problems. Convergence and stability analysis of the implicit RIDC methods can also be found in [9]. Semi-implicit RIDC method was developed in [39]. In 2012, Christlieb, Haynes and Ong [12] combined the time parallel RIDC methods with the space parallel domain decomposition (DD) methods and named it RIDC-DD.

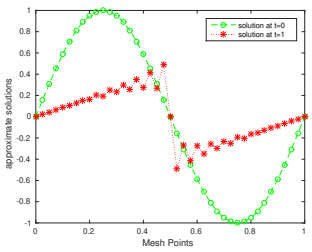
One of the most important features of a RIDC method is its parallel computation of prediction and correction loops. In a recent paper [40], Ong, Haynes and Ladd developed RIDC software aiming at parallel-in-time solution to the initial value problems of the type (1.1). They built an explicit and implicit RIDC library using forward Euler and backward Euler respectively.

There are many partial differential equations (PDEs) whose solutions change very fast over time and space, and these quick variations in the solutions have a negative impact on the efficiency of the methods. In such situations, adaptive mesh methods can be much more efficient than uniform mesh methods. In moving mesh methods, mesh points automatically cluster in the regions where solution changes very rapidly. Adaptive mesh methods for PDEs are classified using the following three categories: h-refinement, p-refinement and r-refinement. In h-refinement strategies, the number of mesh points or the number of elements (in the case of Finite Element Methods (FEMs)) is increased or decreased keeping the order of the numerical methods or the order of the polynomials (in the case of FEM) fixed. The p-refinement strategies change the order of the numerical methods or the order of the polynomials keeping the number of mesh points or the number of elements fixed. In r-refinement technique (moving mesh method), the number of mesh points is kept fixed but their positions are changed or redistributed over time.

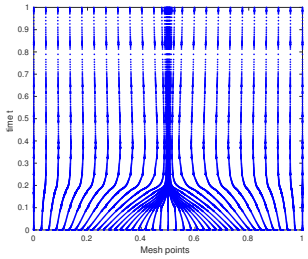
Falgout, Manteuffel, Southworth and Schroder [16] apply a rezoning type moving mesh method to 1D diffusion PDEs. A physical PDE and a moving mesh PDE are discretized and combined together to obtain a coupled system of equations. The coupled system of equations is solved by the parallel-in-time multigrid reduction in time algorithm (MGRIT). In our work, we combine a moving mesh method with the time parallel RIDC method for the solution of one dimensional parabolic PDEs.

To motivate the need for moving mesh methods, consider the non-linear Burgers' equation $u_t = \epsilon u_{xx} - (\frac{u^2}{2})_x,\ x \in [0,1],\ t > 0$, with initial condition $u(x,0) = \sin(2\pi x)$ and boundary conditions $u(0,t) = u(1,t) = 0$. Fig. 1.1(a) shows that the numerical

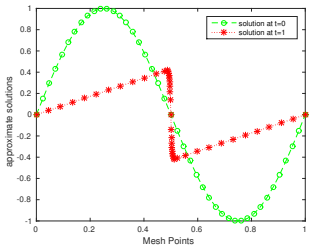
solution using a uniform mesh changes very fast in the region near the point $x = 0.5$. Fig. 1.1(b) illustrates the mesh points clustering towards the point $x = 0.5$ and Fig. 1.1(c) shows the solution, using an adaptive mesh. The rapid change in the solution is efficiently resolved using the moving mesh (see Fig. 1.1(c)).



(a) Solutions at $t = 0$ and $t = 1$ using 41 uniform mesh points.



(b) Mesh movement over time using 41 adaptive mesh points.



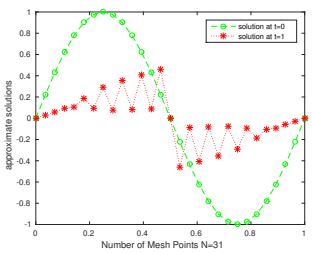
(c) Solutions at $t = 0$ and $t = 1$ on moving mesh using 41 mesh points.

Fig. 1.1: (a) Solutions of Burgers' equation on a uniform mesh with $N = 41$ spatial mesh points and $\epsilon = 10^{-3}$ (b) Mesh clustering towards the point $x = 0.5$ (c) Solutions of Burgers' equation on an adaptive mesh with $N = 41$ spatial mesh points and $\epsilon = 10^{-3}$.

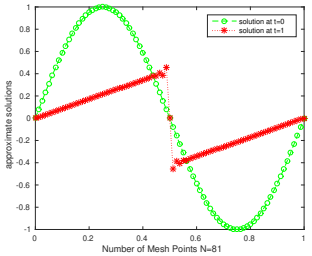
Adaptive mesh techniques can provide better solutions using a smaller number of mesh points compared to the uniform mesh methods. Fig: 1.2 compares the solution of Burgers' equation using a fixed mesh and a moving mesh with different numbers of mesh points. The initial condition and the boundary conditions are the same as in the previous example. Uniform mesh methods typically require more mesh points. It is noticed that a large variation in the solution is still visible using 31 and 81 mesh points in the Fig: 1.2(a) and Fig: 1.2(b) respectively. In contrast, the same solution using an adaptive mesh (see Fig: 1.2(c)) needs only 21 mesh points and is much smoother than the uniform mesh case.

One of the major tasks involved with using a moving mesh method is to formulate an efficient moving mesh partial differential equations (MMPDEs). Details of the derivation of MMPDEs can the found in the book [28] by Huang and Russell. See Chapter 4 for details of how moving mesh solutions are achieved.

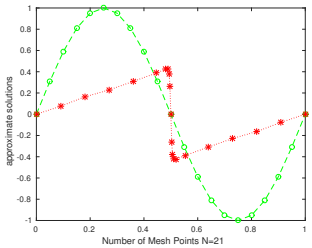
In our thesis, we apply the RIDC method to the system of IVPs which arises in the moving mesh technique for the solution of partial differential equations. We discretize a given PDE and a chosen moving mesh partial differential equation (MMPDE)



(a) Solution at $t = 0$ and $t = 1$ on fixed mesh using 31 mesh points.



(b) Solution at $t = 0$ and $t = 1$ on fixed mesh using 81 mesh points.



(c) Solution at $t = 0$ and $t = 1$ on moving mesh using 21 mesh points.

Fig. 1.2: (a) Solutions of Burgers' equation on the uniform mesh with $N = 31$ and $\epsilon = 10^{-3}$, (b) Solutions of Burgers' equation on the uniform mesh with $N = 81$ and $\epsilon = 10^{-3}$ and (c) Solutions of Burgers' equation on adaptive mesh with $N = 21$ and $\epsilon = 10^{-3}$.

by the (moving) method of lines, and obtain two systems of ODEs. The combination of the two coupled ODE systems gives a $2n \times 2n$ system of ODE of the form $L(t, y)y^{'} = f(t, y)$, where, y consists of the physical solution $u \in \mathbb{R}^n$ and the mesh $x \in \mathbb{R}^n$, and $L(t, y)$ is a state dependent non-singular square matrix. We derive the RIDC formulation for this type of IVP and expand the existing RIDC software library to make it compatible with our derived formulae.

The organization of the remainder of this thesis is as follows. In Chapter 2, we review the formulation of the RIDC method for IVPs with ODEs of the form $y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$, and derive the RIDC formulae for IVPs consisting the ODEs of the forms (i) $Ly^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ where, L is a constant non-singular square matrix, and (ii) $L(t, y)y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$, where $L(t, y)$ is a state dependent non-singular square matrix. In Chapter 3, we describe the existing RIDC software library, and show how it can be expanded to IVPs of the form (i) or (ii) above. In Chapter 4, we review the adaptive moving mesh method including the moving mesh formulation, choosing a monitor function and discretizing PDEs by the moving method of lines. We then apply the RIDC method to the moving method of lines for the solution of non-linear partial differential equations. In Chapter 5, we check the efficiency and the accuracy of the RIDC formulas given in Chapter 2 (on fixed meshes) and Chapter 4 (on moving meshes) by way of several numerical examples. Chapter 6 provides a concluding summary of the thesis.

Chapter 2

Revisionist Integral Deferred Correction methods

A class of time parallel integral deferred correction methods used to solve initial value problems (IVPs) is the family of revisionist integral deferred correction methods (RIDC). The RIDC algorithm is designed so that the prediction and the correction steps can be computed in parallel. More than one processor can be used simultaneously to correct the approximate solutions. The number of processors required is equal to the order of the method. For instance, using four processors a fourth order solution to an IVP can be obtained in approximately the same wall clock time as the first order approximation. This chapter will focus on formulating RIDC formulas for three different types of IVPs along with a detailed discussion on the parallel implementation of the RIDC algorithm.

2.1 RIDC for the ODEs $y'(t) = f(t, y)$

The RIDC method described in [13] presents the approach for the initial value problems of the form

$$y'(t) = f(t, y),\ t \in [0, T],\ y(0) = y_0, \tag{2.1}$$

where $y \in \mathbb{R}^n$.

The key factors needed to construct a RIDC algorithm are the formulation of error equation, and a suitable choice of predictor, corrector and quadrature formula.

This section will talk about these choices.

2.1.1 Error Equation

Let $y(t)$ be the exact solution and $u(t)$ be the approximate solution to the system of IVPs (2.1). Using the approximate solution $u(t)$ in (2.1), we form the residual

$$r(t) = u'(t) - f(t, u). \tag{2.2}$$

The actual error is given by

$$e(t) = y(t) - u(t). \tag{2.3}$$

The derivative of the error equation is then

$$\begin{aligned} e'(t) &= y'(t) - u'(t) \\ &= f(t, y(t)) - f(t, u(t)) - r(t) \\ &= f(t, u(t) + e(t)) - f(t, u(t)) - r(t). \end{aligned} \tag{2.4}$$

Taking the residual term in (2.4) to the left hand side we have

$$e'(t) + r(t) = f(t, u(t) + e(t)) - f(t, u(t)). \tag{2.5}$$

Equation (2.5) can be written in the integral form as

$$\left(e(t) + \int_0^t r(\tau) d\tau \right)' = f(t, u(t) + e(t)) - f(t, u(t)). \tag{2.6}$$

Again from (2.2) we have

$$\int_0^t r(\tau) d\tau = u(t) - u(0) - \int_0^t f(\tau, u(\tau)) d\tau. \tag{2.7}$$

Combining (2.6) and (2.7) we arrive at

$$\left(e(t) + u(t) - \int_0^t f(\tau, u(\tau)) d\tau \right)' = f(t, u(t) + e(t)) - f(t, u(t)). \tag{2.8}$$

Equation (2.6) is used to find the discrete form of the error equation and equation (2.8) is used to find the correction formula for the RIDC method.

2.1.2 Partitions of the Time Domain

We discretize the time domain $[0, T]$ into N intervals so that the uniformly spaced time points are given by

$$t_n = n\Delta t,\ n = 0, 1, \ldots, N, \tag{2.9}$$

where $\Delta t = \frac{T}{N}$ is the uniform step size. The resulting N intervals are further partitioned into J groups so that each group $I_j,\ j = 1, \ldots, J$ contains K $(K >> M)$ subintervals, where M $(= p - 1)$ is the number of corrections required by a p^{th} order RIDC method. The time nodes in each group of intervals are labelled as

$$t_{j,m} = \Big((j-1)K + m)\Big)\Delta t,\ m = 0, 1, \ldots, K,\ j = 1, 2, \ldots, J. \tag{2.10}$$

Each group of intervals

$$I_j = \{t_{j,0}, t_{j,1}, \ldots, t_{j,K}\},\ j = 1, 2, \ldots, J,$$

contains $K + 1$ nodes.

The grouping scheme is illustrated in Fig. 2.1.

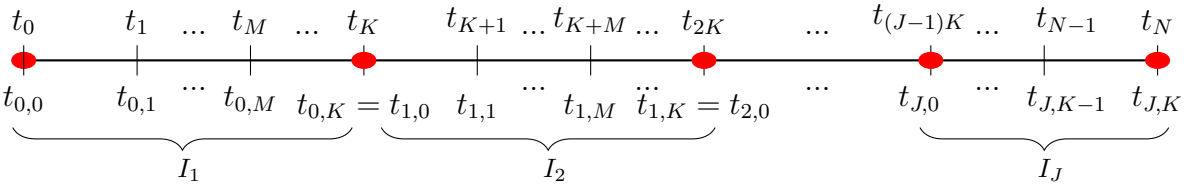


Fig. 2.1: Labeling of nodes used in the RIDC method.

RIDC methods solve over each group of intervals I_j sequentially, starting with the initial group of intervals I_1. The solution at the end of the current group is taken as the initial solution for the next group and it continues until we reach at the last group I_J.

2.1.3 Choice of Predictor and Corrector

The predictors and the correctors used in RIDC methods are chosen based on the nature of the physical problem. For example, an explicit integrator is often sufficient as a predictor and a corrector for the solution of a non-stiff ODE. On the other hand, for the solution of a stiff ODE, a RIDC method requires an implicit integrator as the predictor and the corrector. A lower order integrator is usually chosen for the predictor and the corrector within the RIDC formulation. However, higher order integrators can also be used [11]. In this section we deduce the explicit and the implicit RIDC formulae using the forward Euler and the backward Euler respectively as the integrator.

Discretizing the IVP (2.1) by forward Euler we get the prediction formula of the explicit RIDC method, written as

$$u^{[0]}_{j,m+1} = u^{[0]}_{j,m} + \Delta t f(t_{j,m}, u^{[0]}_{j,m}), \tag{2.11}$$

where $m = 0, 1, \ldots, K-1$. The subscript in $u^{[0]}_{j,m}$ denotes an approximate solution at time $t_{j,m}$ and the superscript $^{[0]}$ indicates that this is approximate solution given by the predictor.

Discretizing the IVP (2.1) by backward Euler we get the prediction formula of the implicit RIDC method, written as

$$u^{[0]}_{j,m+1} = u^{[0]}_{j,m} + \Delta t f(t_{j,m+1}, u^{[0]}_{j,m+1}), \tag{2.12}$$

where $m = 0, 1, \ldots, K-1$.

In each iteration of the correction loop we solve error equations and update the most recently computed approximate solutions. Since $e(0) = 0$, the error equation (2.6) is an IVP. A forward Euler discretization of the error equation (2.6) gives the

approximation

$$e_{j,m+1}^{[l]} = e_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m}, u_{j,m}^{[l-1]} + e_{j,m}^{[l]}) - f(t_{j,m}, u_{j,m}^{[l-1]})\Big) + \int_{t_{j,m}}^{t_{j,m+1}} r(t)dt, \quad (2.13)$$

where $u_{j,m}^{[l-1]}$ denotes the approximate solution of the given IVP at the $(l-1)^{th}$ correction level at time $t_{j,m}$ and $e_{j,m}^{[l]}$ denotes the error in the approximate solution, $u_{j,m}^{[l-1]}$, at time $t_{j,m}$.

Similarly, a backward Euler discretization of the error equation (2.6) gives the approximation

$$e_{j,m+1}^{[l]} = e_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m+1}, u_{j,m+1}^{[l-1]} + e_{j,m+1}^{[l]}) - f(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + \int_{t_{j,m}}^{t_{j,m+1}} r(t)dt. \quad (2.14)$$

The definite integrals in (2.13) and (2.14) can be approximated by any appropriate quadrature rule. The approximate solution can then be updated as

$$u_{j,m+1}^{[l]} = u_{j,m+1}^{[l-1]} + e_{j,m+1}^{[l]},\ l = 1, 2, \ldots, M, \quad (2.15)$$

where $m = 0, 1, \ldots, K-1$.

Discretizing the IVP (2.8) by the forward Euler method and using (2.15) we obtain the correction formula of the explicit RIDC method as

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m}, u_{j,m}^{[l]}) - f(t_{j,m}, u_{j,m}^{[l-1]})\Big) + \int_{t_{j,m}}^{t_{j,m+1}} f(t, u^{[l-1]}(t))dt. \quad (2.16)$$

Discretizing the IVP (2.8) by the backward Euler method and using (2.15) we obtain the correction formula for the implicit RIDC method as

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m+1}, u_{j,m+1}^{[l]}) - f(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + \int_{t_{j,m}}^{t_{j,m+1}} f(t, u^{[l-1]}(t))dt. \quad (2.17)$$

Equations (2.16) and (2.17) give the l^{th} order corrections in terms of the $(l-1)^{th}$ order approximation. The definite integrals in (2.16) and (2.17) can be evaluated by a suitable quadrature rule as discussed next.

2.1.4 Choice of Quadrature Rule

The efficiency of the RIDC method greatly depends on the right choice of quadrature formula. We use the approximation

$$\int_{t_{j,m}}^{t_{j,m+1}} f(t, u^{[l-1]}(t))dt \approx \sum_{i=0}^{M} f(t_{j,i}, u_{j,i}^{[l-1]})W_{mi}, \tag{2.18}$$

where quadrature nodes are the uniform mesh points $t_{j,0}, t_{j,1}, \ldots, t_{j,M}$ and the quadrature weights W_{mi} are the integrals of the M^{th} degree Lagrange interpolating polynomials passing through the points $t_{j,0}, t_{j,1}, \ldots, t_{j,M}$. Since the mesh points $t_{j,0}, t_{j,1}, \ldots, t_{j,M}$ are uniformly spaced and the length of the interval in (2.19) is uniform, the weights computed by (2.19) are identical for all groups. Therefore, quadrature weights are computed only once. We drop the group index j from $t_{j,m}$, and hence we write

$$W_{mi} = \int_{t_m}^{t_{m+1}} \prod_{k=0, k\neq i}^{M} \frac{(t - t_k)}{(t_i - t_k)} dt. \tag{2.19}$$

We can precompute the quadrature weights by transforming the mesh points $t_{j,0}, t_{j,1}, \ldots, t_{j,M}$ to equally spaced points in a fixed interval. This can be done in the following two ways.

Method 1: **Transforming the original mesh points $t_0, t_1, \ldots, t_M$ to uniformly spaced mesh points in $[0, M]$.**

Let $t_0, t_1, \ldots, t_M$ be the uniformly spaced mesh points in the interval $[t_0, t_M]$ with the uniform step size $\Delta t = t_{i+1} - t_i$, $i = 0, \ldots, M-1$. To transform the coordinates t_i, $i = 0, \ldots, M-1$ into the corresponding uniformly spaced coordinates in the interval $[0, M]$ we use the following change of variables

$$t = t_m + \Delta t(s - m).$$

Note that $s = m$ when $t = t_m$ and $s = m+1$ when $t = t_{m+1}$, and $t_k = t_m + \Delta t(k-m)$, for any $k \in [0, M]$.

Therefore,

$$dt = \Delta t ds,$$

and

$$(t - t_k) = (t_m + \Delta t(s - m)) - (t_m + \Delta t(k - m)) = \Delta t(s - k).$$

We also have

$$(t_i - t_k) = (t_m + \Delta t(i - m)) - (t_m + \Delta t(k - m)) = \Delta t(i - k).$$

Therefore the definite integral (2.19) for the quadrature weights can be re-written in the compact form

$$W_{mi} = \Delta t \int_m^{m+1} \prod_{k=0, k\neq i}^{M} \frac{(s-k)}{(i-k)} ds. \tag{2.20}$$

Hence, the explicit RIDC formula (2.16) and the implicit RIDC formula (2.17) take the forms

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m}, u_{j,m}^{[l]}) - f(t_{j,m}, u_{j,m}^{[l-1]})\Big) + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}), \tag{2.21}$$

and

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m+1}, u_{j,m+1}^{[l]}) - f(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}), \tag{2.22}$$

where $m = 0, \ldots, M-1$ and the elements of the integration matrix S are given by

$$S_{mi} = \int_m^{m+1} \prod_{k=0, k\neq i}^{M} \frac{(s-k)}{(i-k)} ds. \tag{2.23}$$

***Method 2*: Transforming the mesh points $t_0, t_1, \ldots, t_M$ to uniformly spaced mesh points in $[0, 1]$.**

Let $t_0, t_1, \ldots, t_M$ be the uniformly spaced mesh points in the interval $[t_0, t_M]$ with the uniform step size $\Delta t = t_{i+1} - t_i$,$i = 0, \ldots, M-1$. To transform the coordinates t_i, $i = 0, \ldots, M-1$ into the corresponding uniformly spaced coordinates in the interval $[0, 1]$, we use the following change of variables

$$t = t_m + M\Delta t(x - x_m),$$

where x_i, $i = 0, \ldots, M$, are the uniformly spaced mesh points in $[0, 1]$ and $m = 0, \ldots, M-1$. Note that $x = x_m$ when $t = t_m$ and $x = x_{m+1}$ when $t = t_{m+1}$, and $t_k = t_m + M\Delta t(k - x_m)$, for any $k \in [0, 1]$.

Hence, we have

$$dt = M\Delta t dx,$$

$$t - t_k = M\Delta t(x - x_k),$$

and

$$t_i - t_k = M\Delta t(x_i - x_k).$$

So, the integrals in (2.19) can be written in the form

$$W_{mi} = M\Delta t \int_{x_m}^{x_{m+1}} \prod_{k=0, k\neq i}^{M} \frac{(x - x_k)}{(x_i - x_k)} dx. \tag{2.24}$$

The correction formula (2.16) for the explicit RIDC can then be expressed as

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m}, u_{j,m}^{[l]}) - f(t_{j,m}, u_{j,m}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}). \tag{2.25}$$

And the correction formula (2.17) for the implicit RIDC can also be expressed as

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m+1}, u_{j,m+1}^{[l]}) - f(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}). \tag{2.26}$$

The elements of the integration matrix S for the correction formulae (2.25) and (2.26) are given by

$$S_{mi} = \int_{x_m}^{x_{m+1}} \prod_{k=0, k\neq i}^{M} \frac{(x - x_k)}{(x_i - x_k)} dx, \tag{2.27}$$

where x_i, $i = 0, \ldots, M$, are the uniformly spaced mesh points in $[0, 1]$ and $m = 0, \ldots, M-1$.

Equation (2.26) can be solved by Newton's method. For the Newton's solver we compute the Jacobian numerically by the finite difference method. This is particularly important when the analytic Jacobian for a given function is quite difficult to compute. In the extended RIDC library (see Chapter 3) we provide a numerical Jacobian routine

inside the Newton solver to approximate the Jacobian.

2.2 RIDC for the ODEs $Ly'(t) = f(t, y)$

In this section, we extend the work of [13] and form the error equation, and construct RIDC formula for the system of ODEs of the form $Ly'(t) = f(t, y)$, where L is a constant invertible matrix. For the selection of the predictor, the corrector and the quadrature formula, we refer the reader to Section 2.1.

2.2.1 Error Equation Formulation

Consider the following IVP consisting of a system of ODEs and initial condition

$$Ly'(t) = g(t, y),\ y(0) = y_0,\ t \in [0, T], \tag{2.28}$$

where $y \in \mathbb{R}^n$ and L is an $n \times n$ constant non-singular matrix. Symbolically, equation (2.28) can be rewritten as

$$y'(t) = L^{-1}g(t, y),\ y(0) = y_0,\ t \in [0, T]. \tag{2.29}$$

Let $y(t)$ be the true solution and $u(t)$ be the approximate solution of the system of IVPs (2.29).

Using the approximate solution $u(t)$ in (2.29), we form the residual

$$r(t) = u'(t) - L^{-1}g(t, u). \tag{2.30}$$

The actual error is given by

$$e(t) = y(t) - u(t). \tag{2.31}$$

The derivative of the error equation (2.31) is given by

$$\begin{aligned} e'(t) &= y'(t) - u'(t) \\ &= L^{-1}g(t, y(t)) - L^{-1}g(t, u(t)) - r(t) \\ &= L^{-1}\Big(g(t, u(t) + e(t)) - g(t, u(t))\Big) - r(t). \end{aligned} \tag{2.32}$$

Taking the residual term in (2.32) to the left hand side we have

$$e'(t) + r(t) = L^{-1}\Big(g(t, u(t) + e(t)) - g(t, u(t))\Big). \tag{2.33}$$

Since $e(0) = 0$, equation (2.33) can be written in the integral form as

$$\left(e(t) + \int_0^t r(\tau)d\tau\right)' = L^{-1}\Big(g(t, u(t) + e(t)) - g(t, u(t))\Big). \tag{2.34}$$

Again from (2.30) we have

$$\int_0^t r(\tau)d\tau = u(t) - u(0) - L^{-1}\int_0^t g(\tau, u(\tau))d\tau. \tag{2.35}$$

Substituting (2.35) into (2.34) and multiplying by L, we obtain the error equation

$$L\left(e(t) + u(t) - L^{-1}\int_0^t g(\tau, u(\tau))d\tau\right)' = \Big(g(t, u(t) + e(t)) - g(t, u(t))\Big). \tag{2.36}$$

Equation (2.34) is used to find the discrete form of the error equation and equation (2.36) is used to find the correction formula for the RIDC method.

2.2.2 The Predictor

Discretizing (2.28) by the forward Euler we get the predictor formula of the explicit RIDC method as

$$L\Big(u^{[0]}_{j,m+1} - u^{[0]}_{j,m}\Big) = \Delta t g(t_{j,m}, u^{[0]}_{j,m}),\ m = 0, \ldots, K-1,\ j = 1, \ldots, J. \tag{2.37}$$

The prediction equation (2.37) be rewritten as

$$\begin{aligned} u^{[0]}_{j,m+1} &= u^{[0]}_{j,m} + \Delta t L^{-1} g(t_{j,m}, u^{[0]}_{j,m}) \\ &= u^{[0]}_{j,m} + \Delta t f(t_{j,m}, u^{[0]}_{j,m}),\ m = 0, \ldots, K-1, \end{aligned} \tag{2.38}$$

where $j = 1, \ldots, J$ is the group index and $f(t_{j,m}, u^{[0]}_{j,m}) = L^{-1} g(t_{j,m}, u^{[0]}_{j,m})$. The quantity $f(t, u)$ can be evaluated by solving the following system of linear equations

$$Lf(t, u) = g(t, u). \tag{2.39}$$

Similarly, a backward Euler discretization of (2.28) gives the prediction formula of the implicit RIDC method as

$$\begin{aligned} u^{[0]}_{j,m+1} &= u^{[0]}_{j,m} + \Delta t L^{-1} g(t_{j,m+1}, u^{[0]}_{j,m+1}) \\ &= u^{[0]}_{j,m} + \Delta t f(t_{j,m+1}, u^{[0]}_{j,m+1}),\ m = 0, \ldots, K-1, \end{aligned} \tag{2.40}$$

where $j = 1, \ldots, J$ is the group index and $f(t_{j,m+1}, u^{[0]}_{j,m+1}) = L^{-1} g(t_{j,m+1}, u^{[0]}_{j,m+1})$. The function value $f(t, u)$ can be evaluated by solving the system of linear equations (2.39). Equation (2.40) can be solved by Newton's method, for example. Note L can be refactorized (for all times) and the factors reused for all linear solves.

2.2.3 The Corrector

Discretizing (2.36) by the forward Euler and using the quadrature formula (2.24) we get the correction formula of the explicit RIDC method written as

$$\begin{aligned} u^{[l]}_{j,m+1} &= u^{[l]}_{j,m} + \Delta t \Big(L^{-1} g(t_{j,m}, u^{[l]}_{j,m}) - L^{-1} g(t_{j,m}, u^{[l-1]}_{j,m}) \Big) + \int_{t_{j,m}}^{t_{j,m+1}} L^{-1} g(t, u^{[l-1]}(t)) dt \\ &= u^{[l]}_{j,m} + \Delta t \Big(L^{-1} g(t_{j,m}, u^{[l]}_{j,m}) - L^{-1} g(t_{j,m}, u^{[l-1]}_{j,m}) \Big) + M \Delta t \sum_{i=0}^{M} S_{mi} L^{-1} g(t_{j,i}, u^{[l-1]}_{j,i}) \\ &= u^{[l]}_{j,m} + \Delta t \Big(f(t_{j,m}, u^{[l]}_{j,m}) - f(t_{j,m}, u^{[l-1]}_{j,m}) \Big) + M \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u^{[l-1]}_{j,i}),\ m = 0, \ldots, M-1 \end{aligned} \tag{2.41}$$

and

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(L^{-1}g(t_{j,m}, u_{j,m}^{[l]}) - L^{-1}g(t_{j,m}, u_{j,m}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{M-1,i} L^{-1} g(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]})$$

$$= u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m}, u_{j,m}^{[l]}) - f(t_{j,m}, u_{j,m}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}),$$
$$m = M, \ldots, K-1, \quad (2.42)$$

where l is the number of corrections required and $j = 1, \ldots, J$ is the group index. For any t and u, $f(t,u) = L^{-1}g(t,u)$ and is evaluated by solving the system of linear equations (2.39). The elements of the integration matrix S are obtained from the formula given in (2.27).

Discretizing (2.36) by the backward Euler we obtain the correction formula of the implicit RIDC method written as

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(L^{-1}g(t_{j,m+1}, u_{j,m+1}^{[l]}) - L^{-1}g(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + \int_{t_{j,m}}^{t_{j,m+1}} L^{-1}g(t, u^{[l-1]}(t))dt$$

$$= u_{j,m}^{[l]} + \Delta t\Big(L^{-1}g(t_{j,m+1}, u_{j,m+1}^{[l]}) - L^{-1}g(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{mi} L^{-1} g(t_{j,i}, u_{j,i}^{[l-1]})$$

$$= u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m+1}, u_{j,m+1}^{[l]}) - f(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}),\ m = 0, \ldots, M-1 \quad (2.43)$$

and

$$u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t\Big(L^{-1}g(t_{j,m+1}, u_{j,m+1}^{[l]}) - L^{-1}g(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{M-1,i} L^{-1} g(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]})$$
$$= u_{j,m}^{[l]} + \Delta t\Big(f(t_{j,m+1}, u_{j,m+1}^{[l]}) - f(t_{j,m+1}, u_{j,m+1}^{[l-1]})\Big) + M\Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}),\ m = M, \ldots, K-1, \quad (2.44)$$

where l is the number of corrections required and $j = 1, \ldots, J$ is the group index. For any t and u, $f(t,u) = L^{-1}g(t,u)$ and is evaluated by solving the system of linear equations (2.39). The elements of the integration matrix S are obtained from formula (2.27). Equations (2.43) and (2.44) can be solved by Newton's method.

2.3 RIDC for the ODEs $L(t,y)y'(t) = f(t,y)$

In this section, we form the error equation and construct RIDC formula for a system of ODEs of the form $L(t,y)y'(t) = f(t,y)$, where $L(t,y)$ is a state dependent non-singular square matrix. For the selection of the predictor, the corrector and the quadrature formula we refer the reader to Section 2.1.

2.3.1 Error Equation Formulation

Consider the following IVP consisting of a system of ODEs and initial condition

$$L(t,y)y'(t) = g(t,y),\ y(0) = y_0,\ t \in [0,T], \quad (2.45)$$

where $y \in \mathbb{R}^n$ and $L(t,y)$ is a state dependent non-singular square matrix. Symbolically, equation (2.45) can be rewritten as

$$y'(t) = L^{-1}(t,y)g(t,y),\ y(0) = y_0,\ t \in [0,T]. \quad (2.46)$$

Let $y(t)$ be the exact solution and $u(t)$ be the approximate solution of the system of IVPs (2.46). Using the approximate solution $u(t)$ in (2.46), we obtain the residual

$$r(t) = u'(t) - L^{-1}(t,u)g(t,u). \tag{2.47}$$

Once again the actual error is

$$e(t) = y(t) - u(t). \tag{2.48}$$

The derivative of the error equation (2.48) is

$$\begin{aligned} e'(t) &= y'(t) - u'(t) \\ &= L^{-1}(t,y)g(t,y(t)) - L^{-1}(t,u)g(t,u(t)) - r(t) \\ &= L^{-1}(t,u(t)+e(t))g(t,u(t)+e(t)) - L^{-1}(t,u)g(t,u(t)) - r(t). \end{aligned} \tag{2.49}$$

Taking the residual term in (2.49) to the left hand side we have

$$e'(t) + r(t) = L^{-1}(t,u(t)+e(t))g(t,u(t)+e(t)) - L^{-1}(t,u)g(t,u(t)). \tag{2.50}$$

Since $e(0) = 0$, equation (2.50) can be written in the integral form as

$$\left(e(t) + \int_0^t r(\tau)d\tau\right)' = L^{-1}(t,u(t)+e(t))g(t,u(t)+e(t)) - L^{-1}(t,u)g(t,u(t)). \tag{2.51}$$

Again from (2.47) we have

$$\int_0^t r(\tau)d\tau = u(t) - u(0) - L^{-1}(t,u)\int_0^t g(\tau,u(\tau))d\tau. \tag{2.52}$$

Substituting the equation (2.52) into (2.51) we have

$$\left(e(t) + u(t) - L^{-1}(t,u)\int_0^t g(\tau,u(\tau))d\tau\right)' = L^{-1}(t,u(t)+e(t))g(t,u(t)+e(t)) - L^{-1}(t,u)g(t,u(t)). \tag{2.53}$$

Equation (2.51) is used to find the discrete form of the error equation and equation (2.53) is used to find the correction formula for the RIDC method. The key difference

between the error equations (2.36) and (2.53) is that the residual term in (2.53) has a multiplicative factor $L^{-1}(t,u)$ which depends on t and u. In error equation (2.36), we have the term LL^{-1} will yields an identity matrix since L is a constant invertible matrix.

2.3.2 The Predictor

Discretizing (2.45) by the forward Euler we have the prediction formula of the explicit RIDC method given by

$$L(t_{j,m},u_{j,m}^{[0]})\Big(u_{j,m+1}^{[0]}-u_{j,m}^{[0]}\Big)=\Delta t g(t_{j,m},u_{j,m}^{[0]}),\ m=0,\ldots,K-1, \tag{2.54}$$

where $j=1,\ldots,J$ is the group index. Equation (2.54) can be rewritten as

$$\begin{aligned} u_{j,m+1}^{[0]} &= u_{j,m}^{[0]}+\Delta t L^{-1}(t_{j,m},u_{j,m}^{[0]})g(t_{j,m},u_{j,m}^{[0]}) \\ &= u_{j,m}^{[0]}+\Delta t f(t_{j,m},u_{j,m}^{[0]}). \end{aligned} \tag{2.55}$$

where $j=1,\ldots,J$ is the group index and $f(t_{j,m},u_{j,m}^{[0]})=L^{-1}(t_{j,m},u_{j,m}^{[0]})g(t_{j,m},u_{j,m}^{[0]})$. The quantity $f(t,u)$ can be evaluated by solving the following system of linear equations

$$L(t,u)f(t,u)=g(t,u). \tag{2.56}$$

Discretizing (2.45) again by backward Euler we get the prediction formula of the implicit RIDC method as

$$L(t_{j,m+1},u_{j,m+1}^{[0]})\Big(u_{j,m+1}^{[0]}-u_{j,m}^{[0]}\Big)=\Delta t g(t_{j,m+1},u_{j,m+1}^{[0]}),\ m=0,\ldots,K-1, \tag{2.57}$$

where $j=1,\ldots,J$ is the group index. Equation (2.57) can be rewritten as

$$\begin{aligned} u_{j,m+1}^{[0]} &= u_{j,m}^{[0]}+\Delta t L^{-1}(t_{j,m+1},u_{j,m+1}^{[0]})g(t_{j,m+1},u_{j,m+1}^{[0]}) \\ &= u_{j,m}^{[0]}+\Delta t f(t_{j,m+1},u_{j,m+1}^{[0]}). \end{aligned} \tag{2.58}$$

where $f(t_{j,m+1},u_{j,m+1}^{[0]})=L^{-1}(t_{j,m+1},u_{j,m+1}^{[0]})g(t_{j,m+1},u_{j,m+1}^{[0]})$ and $j=1,\ldots,J$ is the group index. The quantity $f(t,u)$ can be evaluated by solving the system of linear equations (2.56). Equation (2.58) can be solved by Newton's method.

2.3.3 The Corrector

Discretizing (2.53) by the forward Euler we obtain the correction formula of the explicit RIDC method as

$$\begin{aligned} u_{m+1}^{[l]} &= u_m^{[l]} + \Delta t L^{-1}(t_m, u_m^{[l]}) g(t_m, u_m^{[l]}) - \Delta t L^{-1}(t_m, u_m^{[l-1]}) g(t_{j,m}, u_m^{[l-1]}) + \\ &\qquad \int_{t_{j,m}}^{t_{m+1}} L^{-1}(t, u^{[l-1]}(t)) g(t, u^{[l-1]}(t)) dt \\ &= u_m^{[l]} + \Delta t L^{-1}(t_m, u_m^{[l]}) g(t_m, u_m^{[l]}) - \Delta t L^{-1}(t_m, u_m^{[l-1]}) g(t_{j,m}, u_m^{[l-1]}) + \\ &\qquad M\Delta t \sum_{i=0}^{M} S_{mi} L^{-1}(t_{j,i}, u_{j,i}^{[l-1]}) g(t_{j,i}, u_{j,i}^{[l-1]}) \\ &= u_m^{[l]} + \Delta t f(t_m, u_m^{[l]}) - \Delta t f(t_{j,m}, u_m^{[l-1]}) + \\ &\qquad M\Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}),\ m = 0, \ldots, M-1 \end{aligned} \tag{2.59}$$

and

$$\begin{aligned} u_{j,m+1}^{[l]} &= u_m^{[l]} + \Delta t L^{-1}(t_m, u_m^{[l]}) g(t_m, u_m^{[l]}) - \Delta t L^{-1}(t_m, u_m^{[l-1]}) g(t_{j,m}, u_m^{[l-1]}) + \\ &\qquad M\Delta t \sum_{i=0}^{M} S_{M-1,i} L^{-1}(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}) g(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}) \\ &= u_m^{[l]} + \Delta t f(t_m, u_m^{[l]}) - \Delta t f(t_{j,m}, u_m^{[l-1]}) + \\ &\qquad M\Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}),\ m = M, \ldots, K-1, \end{aligned} \tag{2.60}$$

where the number of corrections required is $l = 1, \ldots, M$, and the group index j has values $j = 1, \ldots, J$. For any t and u, $f(t,u) = L^{-1}(t,u)g(t,u)$ and is evaluated by solving the system of linear equations (2.56). The elements of the integration matrix S are obtained from the formula (2.27).

Discretizing (2.53) by the backward Euler method we obtain the correction

formula of the implicit RIDC method as

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l]}) g(t_{m+1}, u_{m+1}^{[l]}) - \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l-1]}) g(t_{j,m+1}, u_{m+1}^{[l-1]}) + \int_{t_{j,m}}^{t_{m+1}} L^{-1}(t, u^{[l-1]}(t)) g(t, u^{[l-1]}(t)) dt$$

$$= u_m^{[l]} + \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l]}) g(t_{m+1}, u_{m+1}^{[l]}) - \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l-1]}) g(t_{j,m+1}, u_{m+1}^{[l-1]}) + M\Delta t \sum_{i=0}^{M} S_{mi} L^{-1}(t_{j,i}, u_{j,i}^{[l-1]}) g(t_{j,i}, u_{j,i}^{[l-1]})$$

$$= u_m^{[l]} + \Delta t f(t_{m+1}, u_{m+1}^{[l]}) - \Delta t f(t_{j,m+1}, u_{m+1}^{[l-1]}) + M\Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]}),\ m = 0, \ldots, M-1, \quad (2.61)$$

and

$$u_{j,m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l]}) g(t_{m+1}, u_{m+1}^{[l]}) - \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l-1]}) g(t_{j,m+1}, u_{m+1}^{[l-1]}) + M\Delta t \sum_{i=0}^{M} S_{M-1,i} L^{-1}(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}) g(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]})$$

$$= u_m^{[l]} + \Delta t f(t_{m+1}, u_{m+1}^{[l]}) - \Delta t f(t_{j,m+1}, u_{m+1}^{[l-1]}) + M\Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]}),\ m = M, \ldots, K-1, \quad (2.62)$$

where the number of corrections required is $l = 1, \ldots, M$, and the group index j has values $j = 1, \ldots, J$. The elements of the integration matrix S are obtained from the formula given in (2.27). For any t and u, $f(t, u) = L^{-1}(t, u) g(t, u)$ and is evaluated by solving the system of linear equations (2.56). Equations (2.61) and (2.62) can be solved by Newton solver.

2.4 RIDC Implementation Details

This section presents the RIDC method in a step by step algorithm for systems of ODEs $y' = f(t, y)$. For systems of ODEs of $L(t, y)y' = g(t, y)$, evaluation of f requires linear solves involving the matrix $L(t, y)$ and the right hand side function $g(t, y)$. It also discusses the RIDC method with reduced stencils. A parallel implementation of

the RIDC method will be shown at the end of the section.

2.4.1 RIDC Algorithm with Full Stencils

The RIDC algorithm constructed using the forward Euler method is shown in Algorithm 1 (taken from [13]). As stated in Section 2.1.4, the set of quadrature weights over each group of intervals, I_j , $j = 1, \ldots, J$, is uniform. Therefore, the integration matrix S (elements of S are the quadrature weights) is computed using the formula (2.23) outside the time loop and is used over each group of intervals. The process is reset after performing the computation in a group of K sequential intervals and is restarted using the most recent approximate solutions as initial conditions for the successive groups of intervals.

Algorithm 1: RIDC algorithm constructed using the forward Euler with full stencils [13]

Input : time interval $[0, T]$, N=number of intervals, y_0 is the initial condition, p is the order of the method, $M(= p - 1)$ is the number of corrections required to achieve the order p, J is the number of groups, K is the number of subintervals in each group (N should be divisible by K)

`/* Variable Initialization */`

$u_0 \leftarrow y_0$, $\Delta t \leftarrow \frac{T-0}{N}$, $M = p - 1$, $J = \frac{N}{K}$

for $m = 0$ *to* $M - 1$ **do**

  for $i = 0$ *to* M **do**

    $S_{mi} = \int_m^{m+1} \Big(\prod_{k=0, k\neq i}^{M} \frac{t-k}{i-k} \Big) dt$

for $j = 1$ *to* J **do**

  $u_{j,0}^{[0]} = u_{j-1}$

  `/* Prediction loop */`

  for $m = 0$ *to* $(K - 1)$ **do**

    $t_{j,m} = (jK + m)\Delta t$

    $u_{j,m+1}^{[0]} = u_{j,m}^{[0]} + \Delta t f(t_{j,m}, u_{j,m}^{[0]})$

  `/* Correction loop */`

  for $l = 1$ *to* M **do**

    $u_{j,0}^{[l]} = u_{j,0}^{[l-1]}$

    for $m = 0$ *to* $M - 1$ **do**

      $u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t \Big(f(t_{j,m}, u_{j,m+1}^{[l]}) - f(t_{j,m}, u_{j,m+1}^{[l-1]}) \Big) + \Delta t \sum_{i=0}^{M} S_{mi} f(t_{j,i}, u_{j,i}^{[l-1]})$

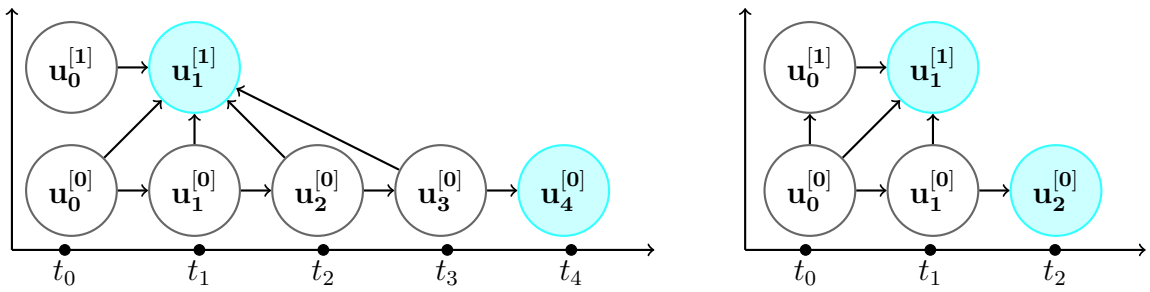
    for $m = M$ *to* $K - 1$ **do**

      $u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t \Big(f(t_{j,m}, u_{j,m+1}^{[l]}) - f(t_{j,m}, u_{j,m+1}^{[l-1]}) \Big) + \Delta t \sum_{i=0}^{M} S_{M-1,i} f(t_{j,m-M+i}, u_{j,m-M+i}^{[l-1]})$

  $u_j = u_{j,K}^{[M]}$

2.4.2 RIDC Algorithm with Reduced Stencils

The definite integral in the RIDC method is approximated by a suitable quadrature rule, and the quadrature weights are computed by integrating the Lagrange interpolating polynomial passing through a set of points. In Algorithm 1, at each level of the correction loop an $M^{th}-$degree Lagrange interpolating polynomial is used to approximate the definite integral. However, it is possible to save computational time by using the $l^{th}-$degree Lagrange polynomial at the l^{th} correction level [13]. RIDC constructed using the $l^{th}-$degree Lagrange polynomial at the l^{th} correction level is shown in Algorithm 2 (again taken from [13]). Using a lower-degree interpolating polynomial reduces the start up cost of the correctors. Fig. 2.2 compares the start up cost of the first corrector of a fourth order RIDC method with reduced stencils and to that with full stencils. Using RIDC with full stencils (Fig. 2.2(a)) the first corrector is lagged behind the predictor by three steps, whereas, if reduced stencils are used (Fig. 2.2(b)) the first corrector starts computing immediately after one step is computed by the predictor.



(a) RIDC with full stencils

(b) RIDC with reduced stencils

Fig. 2.2: The startup of the first corrector of a 4^{th}-order RIDC method (a) with full stencils and (b) with reduced stencils. Input data needed to compute a nodal value is shown by an arrow ($\rightarrow$) pointing towards the node. The colored nodes in each figure indicate that these nodes are computed simultaneously. Using full stencils the first corrector has to wait three steps, whereas for the RIDC with reduced stencils the first corrector has to wait only one step.

Algorithm 2: RIDC algorithm constructed using the forward Euler with reduced stencils [13].

Input : time interval $[0, T]$, N=number of intervals, y_0 is the initial condition, p is the order of the method, $M(= p - 1)$ is the number of correction required to achieve the order p, J is the number of groups, K is the number of subinterval in each group (N should be divisible by K)

`/* Variable Initialization */`

$u_0 \leftarrow y_0$, $\Delta t \leftarrow \frac{T-0}{N}$, $M = p - 1$, $J = \frac{N}{K}$

for $l = 0$ *to* M **do**

  for $m = 0$ *to* $l - 1$ **do**

    for $i = 0$ *to* l **do**

      $S_{mi}^{l} = \int_{m}^{m+1} \Big(\prod_{k=0, k\neq i}^{l} \frac{t-k}{i-k} \Big) dt$

for $j = 1$ *to* J **do**

  $u_{j,0}^{[0]} = u_{j-1}$

  `/* Prediction loop */`

  for $m = 0$ *to* $(K - 1)$ **do**

    $t_{j,m} = (jK + m)\Delta t$

    $u_{j,m+1}^{[0]} = u_{j,m}^{[0]} + \Delta t f(t_{j,m}, u_{j,m}^{[0]})$

  `/* Correction loop */`

  for $l = 1$ *to* M **do**

    $u_{j,0}^{[l]} = u_{j,0}^{[l-1]}$

    for $m = 0$ *to* $l - 1$ **do**

      $u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t \Big(f(t_{j,m}, u_{j,m+1}^{[l]}) - f(t_{j,m}, u_{j,m+1}^{[l-1]}) \Big) +$
      $\Delta t \sum_{i=0}^{l} S_{mi}^{l} f(t_{j,i}, u_{j,i}^{[l-1]})$

    for $m = l$ *to* $K - 1$ **do**

      $u_{j,m+1}^{[l]} = u_{j,m}^{[l]} + \Delta t \Big(f(t_{j,m}, u_{j,m+1}^{[l]}) - f(t_{j,m}, u_{j,m+1}^{[l-1]}) \Big) +$
      $\Delta t \sum_{i=0}^{l} S_{l-1,i}^{l} f(t_{j,m-l+i}, u_{j,m-l+i}^{[l-1]})$

  $u_j = u_{j,K}^{[M]}$

2.4.3 Multi-core Implementation

For a fourth order RIDC method constructed using a forward Euler predictor, the simultaneous computation using four processors is illustrated by Fig. 2.3. The illustration is based on Algorithm 2. It shows that all processors cannot start computing exactly at the same time. Each processor has to wait until necessary input data is available from the previous processors. For example, in Fig. 2.3, the second processor waits until the nodal values $u_0^{[0]}$ and $u_1^{[0]}$ are supplied by the first processor (predictor). While the first processor is computing $u_2^{[0]}$, the second processor starts computing $u_1^{[1]}$. That is, the second processor is always lagged behind from the predictor by one step. The second correction $u_1^{[1]}$ (computed by the third processor) starts while the first processor is computing $u_4^{[0]}$ and the second processor is computing the first correction $u_3^{[1]}$. The third processor starts its job after the completion of the first three steps by the predictor. Similarly, the 3^{rd} correction (by 4^{th} processor) starts when the first six steps are completed by the predictor. In general, each corrector waits $\frac{l(l+1)}{2}$ steps after the predictor starts computing.

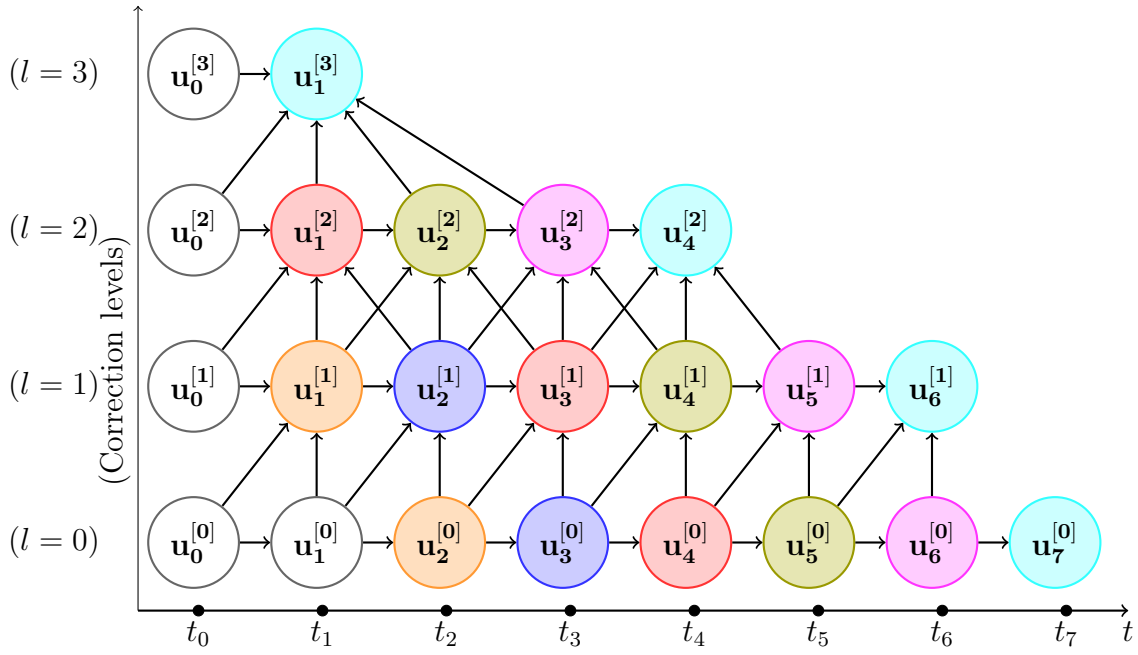


Fig. 2.3: Illustration of the parallel computation of a 4-processor 4^{th} order RIDC method with reduced stencils. The input data needed to compute a nodal value is shown by an arrow ($\rightarrow$) pointing towards the node. Nodes filled with the same color indicate that these nodes are computed simultaneously.

2.4.4 Runtime Analysis

To understand the running time of a RIDC method, we consider here the RIDC method with reduced stencils from [13] (Algorithm 2 in this thesis). We partition a given time domain $[0, T]$ into N equally spaced intervals. A single-processor forward Euler method takes N steps to compute the approximate solution at $t = T$. As described in Section 2.4.3 the RIDC l^{th} corrector (with reduced stencils) is lagged behind by $\frac{l(l+1)}{2}$ steps from the predictor. Therefore, a p-processor RIDC method (p^{th} order) based on Algorithm 2 using forward Euler as a predictor and a corrector will give a p^{th} order solution of an IVP at time $t = T$ in $N + J\frac{M(M+1)}{2}$ steps, where $M = p - 1$ and $J = \frac{N}{K}$.

We assume that the per step cost of the predictor and the corrector are equal and we define the ratio

$$\mu = \frac{\text{number of steps required by the corrector}}{\text{number of steps required by the predictor}} = 1 + \frac{1}{K}\frac{M(M+1)}{2}.$$

Here μ gives a relative measure of the runtime of the predictor and the corrector. Let us suppose that the number of time intervals $n = 100$. If we choose $K = 100$ then $J = 1$, and $\mu = 1.06$ for a 4^{th} order RIDC method with reduced stencils. That is theoretically we expect 6% longer runtime for the 4^{th} order RIDC method (computed using 4-processors) as compared to a single-processor forward Euler method.

If we keep N fixed and set $K = 50$, that is $J = 2$, then the runtime of the 4^{th} order RIDC method with reduced stencils will be 12% (i.e. $\mu = 1.12$) longer than the runtime of a single-processor forward Euler method. That is, the runtime of a RIDC method increases as the value of K becomes smaller.

Chapter 3

The RIDC Software Library and New Extensions

The RIDC software introduced by Ong, Haynes and Ladd [40] was designed to solve IVPs of the form $y'(t) = f(t, y)$, where $y \in \mathbb{R}^n$. In this chapter, we show how to extend the RIDC library to solve systems of IVPs of the forms: (i) $Ly'(t) = g(t, y)$, where $y \in \mathbb{R}^n$ and L is a constant invertible square matrix, and (ii) $L(t, y)y'(t) = g(t, y)$, where $y \in \mathbb{R}^n$, and $L(t, y)$ is an invertible state dependent square matrix. The library implementation process for both forms of the IVPs (i) and (ii) is identical. Hence, we show the implementation of the RIDC library for IVPs of the form (ii) only.

3.1 The Existing RIDC Library

This section is a review of the instructions provided with the RIDC library from [40]. Here, we consider the following initial value problem

$$y'(t) = f(t, y),\ y \in \mathbb{R}^n,\ y(0) = y_0,\ t \in [0, T]. \tag{3.1}$$

3.1.1 Explicit RIDC Library

We recall from the Chapter 2 that the explicit RIDC formula applied to the IVP (3.1) has the form

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t f(t_m, u_m^{[l]}) - \Delta t f(t_m, u_m^{[l-1]}) + \int_{t_m}^{t_{m+1}} f(t, u^{[l-1]}(t))dt. \tag{3.2}$$

For simplicity we have omitted the group index j from $u_m^{[l]}$ by assuming that the formula holds for all groups of intervals. We write here the residual term in integral form. For the computation of the correction formula (3.2) by explicit RIDC library the integral term in (3.2) will be replaced with the quadrature formula of Section 2.1.4.

The library requires a user provided routine called *step* which takes the solution at time t_m as an input and gives a first order approximate solution at time $t_m + \Delta t$, where Δt represents the step size, as output. For the explicit RIDC, using the forward Euler integrator, the user provided *step* routine computes

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t f(t_m, u_m^{[l]}). \tag{3.3}$$

That is, the *step* routine (3.3) takes $u_m^{[l]}$ as an input and returns output $u_{m+1}^{[l]}$.

The explicit RIDC library automatically computes the first two terms of the right hand side of the equation (3.2) by calling the *step* routine, and stores the output as an temporary variable, $w_{m+1}^{[l]}$, given by

$$w_{m+1}^{[l]} = u_m^{[l]} + \Delta t f(t_m, u_m^{[l]}). \tag{3.4}$$

It then computes the last two terms of (3.2) and sums them together with $w_{m+1}^{[l]}$ to get the final output, $u_{m+1}^{[l]}$, given by

$$u_{m+1}^{[l]} = w_{m+1}^{[l]} - \Delta t f(t_m, u_m^{[l-1]}) + \int_{t_m}^{t_{m+1}} f(t, u^{[l-1]}(t))dt. \tag{3.5}$$

This is a *post-processing* step which is shown in the Fig. 3.1. The *post-processing* finally gives the desired corrected solution $u_{m+1}^{[l]}$.

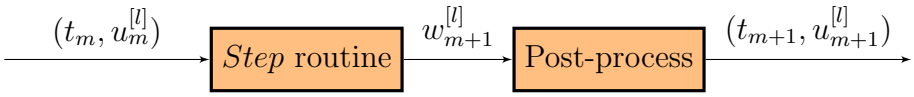


Fig. 3.1: Computation of the correction formula in the explicit RIDC library with post-processing.

3.1.2 Implicit RIDC Library

The subroutine *step* using the backward Euler predictor is able to solve

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t f(t_{m+1}, u_{m+1}^{[l]}) \tag{3.6}$$

by Newton's method or a fixed point iteration for given $u_m^{[l]}$, Δt and t_{m+1}.

We again recall from the Chapter 2 that the implicit RIDC formula applied to the IVP (3.1) takes the form

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t f(t_{m+1}, u_{m+1}^{[l]}) - \Delta t f(t_{m+1}, u_{m+1}^{[l-1]}) + \int_{t_m}^{t_{m+1}} f(t, u^{[l-1]}(t))dt. \tag{3.7}$$

For the computation of the correction formula (3.7) by implicit RIDC library the integral term in (3.7) will be replaced with the quadrature formula of Section 2.1.4.

The first term of the right hand side of (3.7) is known from the previous step and at the l^{th} correction step the 3^{rd} and the 4^{th} terms are known from the $(l-1)^{th}$ correction step. It computes these known terms, sums them together and stores them as a temporary variable, w_m^l, given by

$$w_m^l = u_m^{[l]} - \Delta t f(t_{m+1}, u_{m+1}^{[l-1]}) + \int_{t_m}^{t_{m+1}} f(t, u^{[l-1]}(t))dt. \tag{3.8}$$

Then the correction formula (3.7) requires the solution of

$$u_{m+1}^{[l]} = w_m^l + \Delta t f(t_{m+1}, u_{m+1}^{[l]}). \tag{3.9}$$

Now equation (3.9) is solved by calling the user provided *step* routine which takes w_m^l as an input and yields output, $u_{m+1}^{[l]}$, as the final result. Here the correction terms in (3.8) are computed before calling the *step* routine or the predictor. This is a *pre-processing* step which is shown in Fig. 3.2.

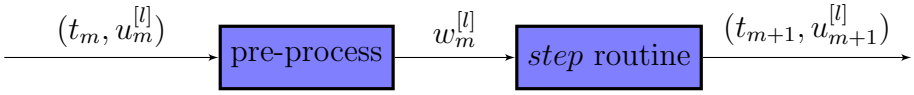


Fig. 3.2: Computation of the correction formula in the implicit RIDC library with pre-processing.

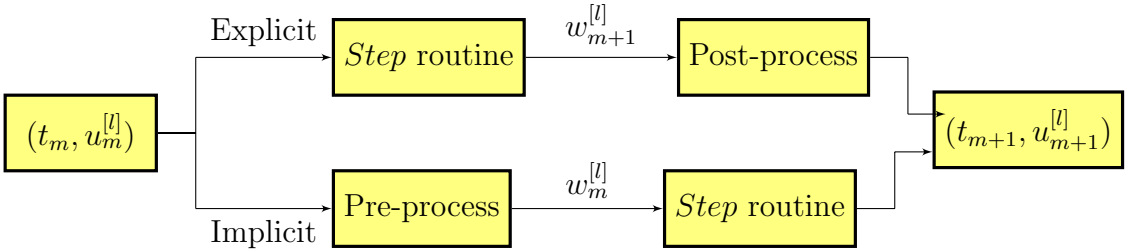


Fig. 3.3: Computation of the correction formulae in the explicit and implicit RIDC library with post-processing and pre-processing respectively.

The different ways of computing of the corrections for the explicit and implicit RIDC methods are illustrated by the combined diagram Fig. 3.3.

3.2 Library Implementation of RIDC for IVPs of the Form $L(t, y)y'(t) = g(t, y)$

3.2.1 Explicit RIDC

For system of ODEs $L(t, y)y'(t) = g(t, y)$, the error equation given in Section 2.3 is

$$\left(e(t) + u(t) - L^{-1}(t, u)\int_0^t g(t, u(\tau))d\tau\right)' = L^{-1}(t, u(t) + e(t))g(t, u(t) + e(t)) - L^{-1}(u)g(t, u(t)). \quad (3.10)$$

Discretizing equation (3.10) by forward Euler we obtain the correction formula for the explicit RIDC method given by

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_m, u_m^{[l]}) g(t_m, u_m^{[l]}) - \Delta t L^{-1}(t_m, u_m^{[l-1]}) g(t_{j,m}, u_m^{[l-1]}) + \int_{t_{j,m}}^{t_{m+1}} L^{-1}(t, u^{[l-1]}(t)) g(t, u^{[l-1]}(t)) dt. \quad (3.11)$$

For the computation of the correction formula (3.11) by explicit RIDC library the integral term in (3.11) will be replaced with the quadrature formula of Section 2.1.4.

The explicit RIDC library [40] can be adapted to (3.11) by providing a user chosen linear solver to compute the values $L^{-1}(t, y)g(t, y)$ at (t, y). Users have to provide the right hand side function $g(t, y)$ of the given IVP and the mass matrix $L(t, y)$. Defining $f(t, y) \equiv L^{-1}(t, y)g(t, y)$ the linear solver solves the system of linear equations

$$L(t, y)f(t, y) = g(t, y), \quad (3.12)$$

to yield the function value $f(t, y)$ at the point (t, y).

In order to make those changes, we provide a subroutine, $gauss(A, B, X)$, a linear solver which takes a square matrix A and a vector B, and returns X as the solution of the system of linear equations $AX = B$. The linear solver *gauss* uses Gaussian elimination. However, any other direct or iterative method can be used. We then modify the subroutine *rhs* given in *explicit.cpp* (*libridc-0.2/examples/explicit/explicit.cpp*) as

```
void  rhs(double t, double *u, double *f)
    {
      // rhs takes inputs t and u, and update f
      g = g(t,u); // RHS of the given IVP to be input by users.
      L = L(t,u); // Mass matrix from given IVP to be input by users.
      n = length(u);
      double *x = new double[n];  // constructor
      gauss(L,g,x); // calling the linear solver
      f = x;
      delete [] x;    // destructor
    }
```

The *step* routine computes the quantity

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_m, u_m^{[l]}) g(t_m, u_m^{[l]}). \tag{3.13}$$

The first two terms of the right hand side of the equation (3.11) are computed by calling the *step* routine and the output is stored as an temporary variable, $w_{m+1}^{[l]}$, given by

$$w_{m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_m, u_m^{[l]}) g(t_m, u_m^{[l]}). \tag{3.14}$$

The rest two terms of the right hand side of (3.11) are computed and are summed together with $w_{m+1}^{[l]}$ to get the final output, $u_{m+1}^{[l]}$, given by

$$u_{m+1}^{[l]} = w_{m+1}^{[l]} - \Delta t L^{-1}(t_m, u_m^{[l-1]}) g(t_{j,m}, u_m^{[l-1]}) + \int_{t_{j,m}}^{t_{m+1}} L^{-1}(t, u^{[l-1]}(t)) g(t, u^{[l-1]}(t)) dt. \tag{3.15}$$

Finally, we add the member function *gauss* in the header file *ridc.h*(*/src/ridc.h*), and compile the script *explicit.cpp* from the directory *libridc-0.2/examples/explicit* using the command *make explicit* as given in [40].

3.2.2 Implicit RIDC

For system of ODEs $L(t, y)y'(t) = g(t, y)$, equation (2.59) of Chapter 2 gives the implicit RIDC formula using backward Euler as

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l]}) g(t_{m+1}, u_{m+1}^{[l]}) - \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l-1]}) g(t_{j,m+1}, u_{m+1}^{[l-1]}) + \int_{t_{j,m}}^{t_{m+1}} L^{-1}(t, u^{[l-1]}(t)) g(t, u^{[l-1]}(t)) dt. \tag{3.16}$$

For the computation of the correction formula (3.16) by implicit RIDC library the integral term in (3.16) will be replaced with the quadrature formula of Section 2.1.4.

In the existing implicit RIDC library (*examples/implicit/implicit.cpp*), the *step* routine computes the solution, $u_{m+1}^{[l]}$, of

$$u_{m+1}^{[l]} = u_m^{[l]} + \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l]}) g(t_{m+1}, u_{m+1}^{[l]}), \tag{3.17}$$

it passes the solution from t_m to t_{m+1} using the backward Euler with a fixed point

iteration method to solve system of equations. We provide a Newton's solver in the *step* routine to solve the system of nonlinear equations. To find the Jacobian matrix for the Newton's method, we provide a subroutine, *jac*, which computes the Jacobian matrix for an arbitrary function, $F(x)$, numerically. Changes also have to be made in the subroutine *rhs* inside *implicit.cpp* (*examples/implicit/implicit.cpp*) to compute function values of the form $L^{-1}(t,y)g(t,y)$ as described in Section 3.2.1.

Now the correction formula (3.16) can easily be computed using the implicit RIDC library, even if the function g is nonlinear. The first term of the right hand side of (3.16) is known from the previous step and at the l^{th} correction step the 3^{rd} and the 4^{th} terms are known from the $(l-1)^{th}$ correction step. The library computes these known terms, sums them together and stores them as a temporary variable, w_m^l, given by

$$w_m^l = u_m^{[l]} - \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l-1]}) g(t_{j,m+1}, u_{m+1}^{[l-1]}) + \int_{t_{j,m}}^{t_{m+1}} L^{-1}(t, u^{[l-1]}(t)) g(t, u^{[l-1]}(t)) dt. \tag{3.18}$$

Then the correction formula (3.16) requires the solution of

$$u_{m+1}^{[l]} = w_m^l + \Delta t L^{-1}(t_{m+1}, u_{m+1}^{[l]}) g(t_{m+1}, u_{m+1}^{[l]}). \tag{3.19}$$

Now equation (3.19) is solved by calling the user provided *step* routine which takes w_m^l as an input and yields the output, $u_{m+1}^{[l]}$, as the final result.

Chapter 4

An Application: A RIDC Moving Method of Lines

The chapter is divided into two sections. The first section deals with a brief discussion on moving mesh technique using the equidistribution principle. In the latter section, we show the semi-discretization of PDEs in space by the moving finite difference method to obtain a system of ODEs. The resulting system of ODEs is then solved by the time parallel implicit RIDC method described in Section 2.3.

4.1 Adaptive Moving Mesh Method

The basic idea behind the moving mesh method is to use mesh points with a variable and time dependent spacing. At each time level mesh points are forced to concentrate in the regions with large solution variations, steep fronts, or oscillations. This section outlines the fundamental ingredients of moving mesh method along with a complete step by step procedure of moving mesh generation.

4.1.1 Choice of Mesh Density Function

The quality of the computed adaptive mesh depends greatly on the right choice of the mesh density function $\rho(x)$. The mesh generated by ρ is concentrated in regions where ρ is large and is scattered in regions where ρ is small. This property of the

adaptive mesh is shown in the Fig. 4.1(b). The most commonly used mesh density functions are the arclength and the curvature based mesh density functions [28].

The arclength mesh density function for a given physical solution u is defined by

$$\rho(x,u,t) = \sqrt{1+|u_x|^2}, \tag{4.1}$$

and the curvature mesh density function is defined by

$$\rho(x,u,t) = (1+|u_{xx}|^2)^{1/4}. \tag{4.2}$$

In practice the derivative term in the arclength mesh density (4.1) is often scaled by some parameter α, giving

$$\rho(x,u,t) = \sqrt{1+\frac{1}{\alpha}|u_x|^2}. \tag{4.3}$$

The scaling factor α reduces the magnitude of ρ in the case when magnitude of the derivative term u_x is very large [41]. A similar scaling can be introduced for the curvature mesh density function.

In this thesis, we use the smoothed arclength mesh density function and the following curvature based mesh density function from [28] which is obtained by minimizing the error between the solution and its interpolant on the equidistributing mesh

$$\rho(x,u,t) = (1+\frac{1}{\alpha}|u_{xx}|^2)^{1/3}, \tag{4.4}$$

where, α in a given interval $[a,b]$ is given by

$$\alpha = \Big[\frac{1}{b-a}\int_a^b |u_{xx}|^{\frac{2}{3}}dx\Big]^3. \tag{4.5}$$

4.1.2 Equidistribution Principle

The equidistribution principle [28] plays a major rule in adaptive moving mesh generation. In one spatial dimension it states that for a given mesh density function, $\rho(x)$, a mesh $\mathcal{T}_h : a = x_1 < x_2 < \cdots < x_N = b$ in the interval $[a,b]$ is to be selected so that the integral value $\int_{x_{i-1}}^{x_i} \rho(x)dx$ is uniform for $i = 2,\ldots,N$, where N is the total

number of mesh points. Mathematically, we write

$$\int_{x_1}^{x_2} \rho(x)\,dx = \cdots = \int_{x_N-1}^{x_N} \rho(x)\,dx. \tag{4.6}$$

That is, the integral value under $\rho(x)$ is uniform in every subinterval $[x_{i-1}, x_i]$ for $i = 2, \ldots, N$.

4.1.3 Adaptive Moving Mesh Generation By the Equidistribution Principle

The integral form of the equidistribution principle given in (4.6) can be rewritten as

$$\int_a^{x_j} \rho(x)\,dx = \frac{j-1}{N-1}\sigma,\ j = 1, \ldots, N, \tag{4.7}$$

where

$$\sigma = \int_a^b \rho(x)\,dx. \tag{4.8}$$

Let us consider a coordinate transformation $x = x(\xi) : [0,1] \to [a,b]$, defined so that

$$x_j = x(\xi_j)\ j = 1, \ldots, N, \tag{4.9}$$

where

$$\xi_j = \frac{j-1}{N-1},\ j = 1, \ldots, N, \tag{4.10}$$

is a uniform mesh on $[0,1]$. Then the equation (4.6) can be rewritten as

$$\int_a^{x(\xi_j)} \rho(x)\,dx = \sigma\xi_j,\ j = 1, \ldots, N. \tag{4.11}$$

In a continuous form, equation (4.11) can be written as

$$\int_a^{x(\xi)} \rho(x)\,dx = \sigma\xi,\ \forall \xi \in [0,1]. \tag{4.12}$$

Differentiating (4.12) with respect to ξ yields

$$\rho(x)\frac{\partial x}{\partial \xi} = \sigma. \tag{4.13}$$

Differentiating (4.13) with respect to ξ we have

$$\frac{\partial}{\partial \xi}\left(\rho(x)\frac{\partial x}{\partial \xi}\right) = 0. \tag{4.14}$$

The solution of the quasi-linear second-order differential equation (4.14) subject to the boundary conditions

$$x(0) = a \text{ and } x(1) = b, \tag{4.15}$$

gives the equidistributing mesh $x_i(\xi)$, $i = 1, \ldots, N$. Several other moving mesh PDEs are shown in Section 4.2.1. In order to numerically solve (4.14) together with the boundary conditions (4.15), we discretize (4.14) using central finite differences on a uniform computational mesh ξ_j, $j = 1, \ldots, N$. This gives for $j = 2, \ldots, N-1$,

$$\frac{2}{\xi_{j+1} - \xi_{j-1}}\Big(\frac{\rho(x_{j+1}) + \rho(x_j)}{2}\frac{(x_{j+1} - x_j)}{\xi_{j+1} - \xi_j} - \frac{\rho(x_j) + \rho(x_{j-1})}{2}\frac{(x_j - x_{j-1})}{\xi_j - \xi_{j-1}}\Big) = 0, \tag{4.16}$$

with boundary conditions

$$x_1 = a \text{ and } x_N = b. \tag{4.17}$$

The mesh density function $\rho(x)$ is often nonlinear and the system of nonlinear equations (4.16) together with the boundary conditions (4.17) can be solved by Newton's method.

Alternatively, it can be solved by the linearization of the system of nonlinear equations. We assume that the mesh density function at the current iteration, ρ^n, is fixed. Then the linearized form of the nonlinear system (4.16) takes the form

$$\begin{aligned}\frac{2}{\xi_{j+1} - \xi_{j-1}}\Big(&\frac{\rho(x_{j+1}^{(n)}) + \rho(x_j^{(n)})}{2}\frac{(x_{j+1}^{(n+1)} - x_j^{(n+1)})}{\xi_{j+1} - \xi_j} \\ &- \frac{\rho(x_j^{(n)}) + \rho(x_{j-1}^{(n)})}{2}\frac{(x_j^{(n+1)} - x_{j-1}^{(n+1)})}{\xi_j - \xi_{j-1}}\Big) = 0.\end{aligned} \tag{4.18}$$

And the boundary conditions (4.17) become

$$x_1^{(n+1)} = a \text{ and } x_N^{(n+1)} = b. \tag{4.19}$$

Now the system of linear equations (4.18) coupled with the boundary conditions (4.19)

is equivalent to the system of linear equations

$$AX = F, \tag{4.20}$$

where A is an $N \times N$ tridiagonal matrix whose entries, for $j = 2, \ldots, N-1$, are given by

$$A(j, j-1) = \frac{1}{\xi_{j+1} - \xi_{j-1}} . \frac{1}{\xi_j - \xi_{j-1}} \Big(\rho(x_j^{(n)}) + \rho(x_{j-1}^{(n)}) \Big),$$

$$A(j, j+1) = \frac{1}{\xi_{j+1} - \xi_{j-1}} . \frac{1}{\xi_{j+1} - \xi_j} \Big(\rho(x_{j+1}^{(n)}) + \rho(x_j^{(n)}) \Big),$$

and

$$A(j,j) = -\frac{1}{\xi_{j+1} - \xi_{j-1}} . \frac{1}{\xi_j - \xi_{j-1}} \Big(\rho(x_j^{(n)}) + \rho(x_{j-1}^{(n)}) \Big) - \frac{1}{\xi_{j+1} - \xi_{j-1}} . \frac{1}{\xi_{j+1} - \xi_j} \Big(\rho(x_{j+1}^{(n)}) + \rho(x_j^{(n)}) \Big),$$

with

$$A(1,1) = 1 \text{ and } A(N,N) = 1.$$

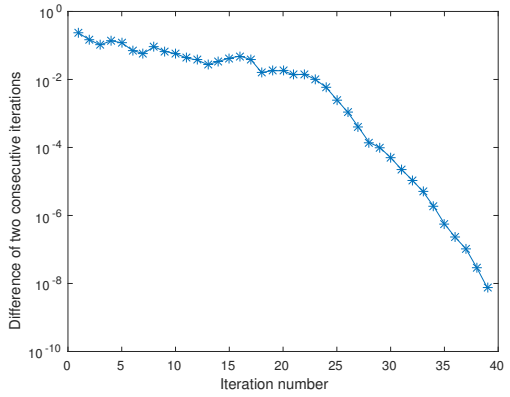
The right hand side function F is an $N \times 1$ matrix whose entries are equal to zero, for $j = 2, \ldots, N-1$, with the fixed boundary values $F(1) = 1$ and $F(N) = 1$.

Example 4.1: Consider the following mesh density function

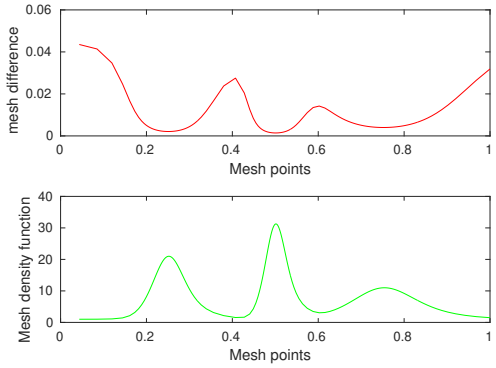
$$\rho(x) = 1 + 20(1 - \tanh^2(20(x - 0.25))) + 30(1 - \tanh^2(30(x - 0.5))) + 10(1 - \tanh^2(10(x - 0.75))), \ \forall x \in [0, 1]. \tag{4.21}$$

Using the mesh density function (4.21) the system of linear equations in (4.18) together with the boundary conditions (4.19) is solved for x by Gaussian elimination. The convergence of iteration (4.18) with a stopping tolerance $tol = 10^{-8}$ is shown by a Fig: 4.1(a). The mesh differences are compared with the mesh density function values $\rho(x_j)$, $j = 1, \ldots, N$, in Fig: 4.1(b). The upper red curve in Fig: 4.1(b) represents the consecutive mesh differences and the lower green curve indicates the function values ρ at the corresponding mesh points x_j, $j = 1 \ldots, N$. We see that the mesh density function ρ takes larger value where the mesh difference is small (i.e near $x = 0.21$ and $x = 0.45$) and ρ takes smaller values where the mesh difference is large (i.e near the points $x = 0$, $x = 0.4$, and $x = 0.4$). The method is repeated using different numbers

of mesh points (N), and the number of iterations needed for convergence of iteration (4.18) is shown in Table 4.1.



(a) The infinity norm of the difference of two consecutive iterations is plotted against the iteration number.



(b) Consecutive mesh differences are compared with the corresponding function values of $\rho(x)$ at the new mesh points.

Fig. 4.1: Using the mesh density function (4.21) the system of linear equations (4.18) along with the boundary conditions (4.19) is solved for x iteratively with an iteration tolerance of $tol = 10^{-8}$ and 160 mesh points in the interval $[0, 1]$. (a) $\max_j \|x_j^{(n+1)} - x_j^{(n)}\|$ is plotted against the number of iterations required to achieve $\max_j \|x_j^{(n+1)} - x_j^{(n)}\| < 10^{-8}$. (b) Mesh spacing $|x_{j+1} - x_j|$ (upper figure) in the newly generated mesh is compared with the corresponding values of the density function $\rho(x_j)$ (lower figure).

Table 4.1: The number of iterations required to achieve $\max_j \|x_j^{(n+1)} - x_j^{(n)}\| < 10^{-8}$ for the Example 4.1 with different numbers of mesh points N. The convergence rate is faster for larger values of N. For the values of N up to 50 the method does not converge (N.conv).

N	10	50	80	161	361
Iteration	N.Conv.	N.Conv.	89	39	40

4.2 Discretization of PDEs on a Moving Mesh

In this section, we introduce moving mesh PDEs. We then apply the (moving) method of lines to a physical PDE and a moving mesh PDE. This semi-discretization results in two $N \times N$ systems of ODEs. The combination of these systems yields a $2N \times 2N$

coupled system of ODEs involving the derivatives of physical solution u and the mesh x.

4.2.1 Moving Mesh PDEs

In the time dependent coordinate case we introduce a time dependent coordinate transformation

$$x = x(\xi, t),\ \xi \in [0, 1], \tag{4.22}$$

which satisfies boundary values

$$x(0, t) = 0 \text{ and } x(1, t) = 1. \tag{4.23}$$

The equidistribution principle states

$$\int_0^{x(\xi,t)} \rho(x, t)\, dx = \xi\sigma(t), \tag{4.24}$$

where

$$\sigma(t) = \int_0^1 \rho(x, t)dx. \tag{4.25}$$

Differentiating equation (4.24) with respect to ξ, we have

$$\rho(x, t)\frac{\partial x(\xi, t)}{\partial \xi} = \sigma(t). \tag{4.26}$$

Differentiating (4.26) again with respect to ξ, we have

$$\frac{\partial}{\partial \xi}\left(\rho(x, t)\frac{\partial x(\xi, t)}{\partial \xi}\right) = 0. \tag{4.27}$$

Equation (4.27) does not contain the mesh speed $\dot{x}(\xi, t)$ and is called a quasi-static equidistribution principle (QSEP). The mesh speed $\dot{x}(\xi, t)$ is particularly important for regularizing the mesh movement. By introducing a time differentiation to the above QSEP Haung, Ren and Russell [27] constructed the following moving mesh

partial differential equations (MMPDEs);

$$\text{MMPDE1:} \quad \frac{\partial}{\partial \xi}\left(\rho \frac{\partial x_t}{\partial \xi}\right) + \frac{\partial}{\partial \xi}\left(\frac{\partial \rho}{\partial \xi} x_t\right) = -\frac{\partial}{\partial \xi}\left(\frac{\partial \rho}{\partial t}\frac{\partial x}{\partial \xi}\right),$$

$$\text{MMPDE2:} \quad \frac{\partial}{\partial \xi}\left(\rho \frac{\partial x_t}{\partial \xi}\right) + \frac{\partial}{\partial \xi}\left(\frac{\partial \rho}{\partial \xi} x_t\right) = -\frac{\partial}{\partial \xi}\left(\frac{\partial \rho}{\partial t}\frac{\partial x}{\partial \xi}\right) - \frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right),$$

$$\text{MMPDE3:} \quad \frac{\partial^2}{\partial \xi^2}\left(\rho x_t\right) = -\frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right),$$

$$\text{MMPDE4:} \quad \frac{\partial}{\partial \xi}\left(\rho \frac{\partial x_t}{\partial \xi}\right) = -\frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right),$$

$$\text{MMPDE5:} \quad x_t = -\frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right),$$

$$\text{Modified MMPDE5:} \quad x_t = -\frac{1}{\tau \rho}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right)$$

and

$$\text{MMPDE6:} \quad \frac{\partial^2 x_t}{\partial \xi^2} = -\frac{1}{\tau}\frac{\partial}{\partial \xi}\left(\rho \frac{\partial x}{\partial \xi}\right).$$

Derivations and the applications of the above mentioned MMPDEs can be found in [27, 28]. In this thesis, we use MMPDE6 for our mesh movement.

4.2.2 Moving Method of Lines

As an example of a nonlinear parabolic partial differential equation, we consider the one dimensional Burgers' equation

$$u_t = \epsilon u_{xx} - \left(\frac{u^2}{2}\right)_x, \; x \in [0,1], \; t > 0, \tag{4.28}$$

along with the boundary conditions

$$u(0,t) = 0 \text{ and } u(1,t) = 0. \tag{4.29}$$

We consider the time dependent coordinate transformation from the computational domain $\Omega_c \equiv [0,1]$ to the physical domain $\Omega \equiv [0,1]$

$$x = x(\xi, t) : \Omega_c \to \Omega. \tag{4.30}$$

Then the coordinates of the new mesh points are given by

$$x_j(t) = x(\xi_j, t), \quad j = 1, \ldots, N, \tag{4.31}$$

where

$$\xi_j = \frac{j-1}{N-1}, \quad j = 1, \ldots \quad , N \tag{4.32}$$

is a uniform mesh in the fixed computational domain Ω_c.

The solution of (4.28) in the transformed domain takes the form

$$\hat{u}(\xi, t) = u(x(\xi, t), t). \tag{4.33}$$

Since $x = x(\xi, t)$, by the chain rule we have

$$\hat{u}_\xi = \ \frac{d}{d\xi}(\hat{u}) = \frac{du}{dx}\frac{dx}{d\xi} = u_x x_\xi. \tag{4.34}$$

This implies that

$$u_x = \frac{\hat{u}_\xi}{x_\xi}, \tag{4.35}$$

and

$$u_{xx} = (\frac{\hat{u}_\xi}{x_\xi})_x = \frac{(\frac{\hat{u}_\xi}{x_\xi})_\xi}{x_\xi}. \tag{4.36}$$

We observe

$$\hat{u}_t = \ \frac{d}{dt}(\hat{u}) = u_t + u_x x_t. \tag{4.37}$$

This implies that

$$\begin{aligned} u_t &= \hat{u}_t - u_x x_t \\ &= \hat{u}_t - u_x x_t \\ &= \hat{u}_t - \frac{\hat{u}_\xi}{x_\xi} x_t. \end{aligned} \tag{4.38}$$

$$\Big(\frac{u^2}{2}\Big)_x = \frac{1}{x_\xi}\Big(\frac{u^2}{2}\Big)_\xi. \tag{4.39}$$

Using equations (4.33)-(4.39) in (4.28), we have

$$\hat{u}_t - \frac{\hat{u}_\xi}{x_\xi} x_t = \frac{\epsilon}{x_\xi}\Big(\frac{\hat{u}_\xi}{x_\xi}\Big)_\xi - \frac{1}{x_\xi}\Big(\frac{\hat{u}^2}{2}\Big)_\xi. \tag{4.40}$$

Using central finite differences in the computational domain, Ω_c, we have

$$x_\xi = \frac{x_{j+1} - x_{j-1}}{2\Delta\xi}, \tag{4.41}$$

$$\Big(\frac{\hat{u}^2}{2}\Big)_\xi = \frac{1}{4}\frac{u_{j+1}^2 - u_{j-1}^2}{\Delta\xi}, \tag{4.42}$$

and

$$\Big(\frac{\hat{u}_\xi}{x_\xi}\Big)_\xi = \frac{2}{2\Delta\xi}\Big[\frac{u_{j+1} - u_j}{x_{j+1} - x_j} - \frac{u_j - u_{j-1}}{x_j - x_{j-1}}\Big], \tag{4.43}$$

where

$$\Delta\xi = \frac{1}{N-1}. \tag{4.44}$$

Using equations (4.41)-(4.43) in (4.40), we have

$$\frac{du_j}{dt} - \frac{(u_{j+1} - u_{j-1})}{(x_{j+1} - x_{j-1})}\frac{dx_j}{dt} = \frac{2\epsilon}{(x_{j+1} - x_{j-1})}\Big[\frac{u_{j+1} - u_j}{x_{j+1} - x_j} - \frac{u_j - u_{j-1}}{x_j - x_{j-1}}\Big] - \frac{1}{2}\frac{(u_{j+1}^2 - u_{j-1}^2)}{(x_{j+1} - x_{j-1})},\ j = 2, \ldots, N-1. \tag{4.45}$$

The boundary conditions (4.29) can be rewritten as

$$\frac{du_1}{dt} = 0 \text{ and } \frac{du_N}{dt} = 0. \tag{4.46}$$

Here, $u_j(t)$ is the approximation to $\hat{u}(\xi_j, t)$ and u_j will be approximated based on the mesh points x_j. The mesh points x_j can be determined by solving, for example, MMPDE6 from Section 4.2.1.

$$\frac{\partial^2 x_t}{\partial\xi^2} = -\frac{1}{\tau}\frac{\partial}{\partial\xi}\Big(\rho\frac{\partial x}{\partial\xi}\Big), \tag{4.47}$$

together with the boundary conditions

$$x(0,t) = 0 \text{ and } x(1,t) = 1, \tag{4.48}$$

where, ρ is the mesh density function and $\tau > 0$ is a user-defined parameter which controls the mesh movement due to the change in $\rho(x,t)$. The mesh moves faster when τ is small and the mesh movement becomes slow when τ is large.

The semi-discretization of (4.47) in the computational domain, Ω_c, gives

$$\frac{(x_t)_{j-1} - 2(x_t)_j + (x_t)_{j+1}}{\Delta\xi^2} = -\frac{1}{\tau}\frac{2}{2\Delta\xi}\Big[\frac{\rho_{j+1}+\rho_j}{2}\frac{(x_{j+1}-x_j)}{\Delta\xi} - \frac{\rho_j+\rho_{j-1}}{2}\frac{(x_j-x_{j-1})}{\Delta\xi}\Big]. \tag{4.49}$$

Simplifying (4.49), we have

$$\frac{dx_{j-1}}{dt} - 2\frac{dx_j}{dt} + \frac{dx_{j+1}}{dt} = -\frac{1}{\tau}\Big[\frac{\rho_{j+1}+\rho_j}{2}(x_{j+1}-x_j) - \frac{\rho_j+\rho_{j-1}}{2}(x_j-x_{j-1})\Big], \tag{4.50}$$

and rewriting the boundary conditions (4.48) gives

$$\frac{dx_1}{dt} = 0 \text{ and } \frac{dx_N}{dt} = 0. \tag{4.51}$$

Equations (4.45) and (4.50) together with the boundary conditions (4.46) and (4.51) form a coupled system of $2N$ ODEs for the physical solution $u_j(t)$ and the mesh $x_j(t)$, $j = 1, \ldots, N$. The coupled ODE system can be written in the the following mass matrix form

$$L(t,y)y' = g(t,y), \tag{4.52}$$

where

$$y = [u_1(t), \ldots, u_N(t), x_1(t), \ldots, x_N(t)]^T,\ y' = \frac{dy}{dt}, \tag{4.53}$$

$g(t,y)$ is the right hand side function of the coupled system of ODEs and $L(t,y)$ is a $2N \times 2N$ matrix given by

$$L(t,y) = \left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right),$$

where M_1, M_2, M_3 are sparse diagonal matrices, and M_4 is a sparse tridiagonal matrix. The matrices M_1, M_3 and M_4 are given by

$$M_1 = \begin{pmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{pmatrix},\ M_3 = \begin{pmatrix} 0 & & \\ & \ddots & \\ & & 0 \end{pmatrix} \text{ and } M_4 = \begin{pmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}.$$

M_2 has diagonal elements $-\frac{u_{j+1}-u_{j-1}}{x_{j+1}-x_{j-1}}$ for $j = 2, \ldots, N-1$, and $M_2(1,1) = M_2(N,N) = 1$. The system of ODEs (4.52) can be solved by the RIDC-BE formula of Section 2.3. To do this we need to choose a mesh density function first. In this thesis, we use the curvature and arclength mesh density functions.

The scaled curvature based mesh density function [28] defined by

$$\rho_j = \Big(1 + \frac{1}{\alpha_h}|u_{xx,j}|^2\Big)^{\frac{1}{3}}, \tag{4.54}$$

where

$$\alpha_h = \max\Big(1, \Big[\sum_{j=2}^{N} \frac{1}{2}(x_j - x_{j-1})\Big(|u_{xx,j}|^{\frac{2}{3}} + |u_{xx,j-1}|^{\frac{2}{3}}\Big)\Big]^3\Big), \tag{4.55}$$

and the spatial derivatives are approximated by

$$u_{xx,j} = \frac{2}{(x_{j+1} - x_{j-1})}\Big[\frac{u_{j+1} - u_j}{x_{j+1} - x_j} - \frac{u_j - u_{j-1}}{x_j - x_{j-1}}\Big],\ j = 2, \ldots, N-1. \tag{4.56}$$

The values at the end points are given by

$$u_{xx,1} = \frac{2\Big[(x_2 - x_1)(u_3 - u_1) - (x_3 - x_1)(u_2 - u_1)\Big]}{(x_3 - x_1)(x_2 - x_1)(x_3 - x_2)}, \tag{4.57}$$

and

$$u_{xx,N} = \frac{2\Big[(x_{N-1} - x_N)(u_{N-2} - u_N) - (x_{N-2} - x_N)(u_{N-1} - u_N)\Big]}{(x_{N-2} - x_N)(x_{N-1} - x_N)(x_{N-2} - x_{N-1})}. \tag{4.58}$$

In the case when u is not smooth, it is convenient to use a smoothed mesh density function. An effective smoothing technique, see [28], uses

$$\rho_j = \frac{1}{4}\rho_{j-1} + \frac{1}{2}\rho_j + \frac{1}{4}\rho_{j+1},\ j = 2, \ldots, N-1,$$

with

$$\rho_1 = \frac{1}{2}\rho_1 + \frac{1}{2}\rho_2,$$

and

$$\rho_N = \frac{1}{2}\rho_{N-1} + \frac{1}{2}\rho_N.$$

We will also test the arclength mesh density function

$$\rho(x,u,t) = \sqrt{1+|u_x|^2}. \tag{4.59}$$

Discretization of (4.59) in the computational domain gives

$$\rho_i = \sqrt{1+\Big(\frac{u_{i+1}-u_{i-1}}{x_{i+1}-x_{i-1}}\Big)^2},\ i = 2,\ldots,N-1. \tag{4.60}$$

A smoothing scheme [26] for the arclength mesh density function (4.60) is given by

$$\tilde{\rho}_i = \sqrt{\sum_{k=i-p}^{i+p} (\rho_k)^2 \Big(\frac{\gamma}{1+\gamma}\Big)^{|k-i|} \Big/ \sum_{k=i-p}^{i+p} \Big(\frac{\gamma}{1+\gamma}\Big)^{|k-i|}}, \tag{4.61}$$

where γ is a positive constant called smoothing parameter and p is a nonnegative integer called smoothing index. The choice of the parameters γ and p can be found in [26].

Chapter 5

Numerical Results

In this chapter, we provide a number of examples illustrating the efficiency and accuracy of the RIDC formulae derived in Chapter 2 and Chapter 4. The results are computed using the enhanced library described in Chapter 3. We begin by showing the techniques by which the order of the accuracy of the methods are calculated.

5.1 Estimating the Order of Convergence

In this thesis, we use two types of methods to verify the order of accuracy of the RIDC methods. The first method is used for the cases where true solution of a given IVP is known and the second method is used for the cases where true solution of the IVP is not known.

5.1.1 Method-1: (When True Solution is Known)

Let U be the exact solution to a given scalar IVP and $\bar{U}(h)$ be its numerical solution at a fixed time T. The error obtained with a numerical method with step size h is given by

$$E(h) \equiv U - \bar{U}(h). \tag{5.1}$$

For a pth order method, we expect

$$E(h) = Ch^p + \mathcal{O}(h^{p+1}), \text{ as } h \to 0. \tag{5.2}$$

When h is sufficiently small, we have

$$E(h) \approx Ch^p. \tag{5.3}$$

When step size h is halved we expect

$$E(h/2) \approx C(h/2)^p. \tag{5.4}$$

Combining (5.3) and (5.4), we see

$$\frac{E(h)}{E(h/2)} \approx 2^p. \tag{5.5}$$

Hence, p can be estimated by

$$p \approx \log_2 \Big(\frac{E(h)}{E(h/2)}\Big). \tag{5.6}$$

5.1.2 Method-2: (When True Solution is not Known)

When the exact solution is not known the step size h is halved successively and the order of accuracy is computed by taking the ratios of differences between two computed solutions for successive h. Using the equations (5.1) and (5.2), we have

$$\begin{aligned} \bar{E}(h) &\equiv U(h) - U(h/2). \\ &= (U(h) - U) - (U(h/2) - U). \\ &= Ch^p - \Big(\frac{h}{2}\Big)^p + \mathcal{O}(h^{p+1}),\ h \to 0. \\ &= C\Big(1 - \frac{1}{2^p}\Big)h^p + \mathcal{O}(h^{p+1}),\ h \to 0. \end{aligned} \tag{5.7}$$

Hence, when h is sufficiently small, we have

$$\bar{E}(h) \approx C\Big(1 - \frac{1}{2^p}\Big)h^p. \tag{5.8}$$

By the similar argument, we get

$$\begin{aligned} \bar{E}(h/2) &= U(h/2) - U(h/4). \\ &\approx C\Big(1 - \frac{1}{2^p}\Big)\frac{h^p}{2^p}. \end{aligned} \tag{5.9}$$

Taking the ratio of (5.8) and (5.9), we have

$$\frac{\bar{E}(h)}{\bar{E}(h/2)} \approx 2^p. \tag{5.10}$$

Hence, p can be estimated by

$$p \approx \log_2 \Big(\frac{\bar{E}(h)}{\bar{E}(h/2)}\Big). \tag{5.11}$$

5.2 Simple IVPs and a 1D Heat Equation on a Uniform Mesh

The order of accuracy of the explicit and the implicit RIDC methods developed in this thesis using forward euler and backward Euler predictors respectively, are tested here for several initial value problems. We summarize our observations and conclusions for all of these examples in Section 5.4.

Example 5.1: Let us consider the following IVP

$$y'(t) = y(t),\ y(0) = 1,\ t \in [0, 1], \tag{5.12}$$

with the exact solution

$$y(t) = e^t.$$

The explicit RIDC (RIDC-FE) method and the implicit RIDC (RIDC-BE) method of Section 2.1 are applied to the IVP (5.12). The methods are tested for different time steps. The order of accuracy in both cases are computed by *Method 1* given in Section 5.1.1. The errors and the orders of accuracy of RIDC-FE and RIDC-BE are shown in Table 5.1 and Table 5.2 respectively.

Table 5.1: Errors and orders of accuracy of RIDC-FE method applied to the IVP (5.12). The number of subintervals in each group is $K = 20$ and the step size is $\Delta t = 0.01$.

Step size	RIDC-FE-1		RIDC-FE-2		RIDC-FE-3		RIDC-FE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	1.973E-2	—	7.517E-5	—	4.37E-7	—	2.207E-9	—
$\Delta t/2$	9.912E-3	0.9931	1.771E-5	2.086	4.908E-8	3.154	1.162E-10	4.248
$\Delta t/4$	4.968E-3	0.9965	4.289E-6	2.046	5.791E-9	3.083	6.591E-12	4.140
$\Delta t/8$	2.487E-3	0.9983	1.055E-6	2.024	7.023E-10	3.044	3.921E-13	4.071

Table 5.2: Errors and orders of accuracy of RIDC-BE method applied to the IVP (5.12). The number of subintervals in each group is $K = 20$ and the step size is $\Delta t = 0.01$.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.01372	—	5.293E-5	—	5.223E-7	—	1.672E-9	—
$\Delta t/2$	0.006827	1.007	1.227E-5	2.109	6.011E-8	3.119	8.378E-11	4.319
$\Delta t/4$	0.003406	1.003	2.949E-6	2.057	7.203E-9	3.061	4.631E-12	4.177
$\Delta t/8$	0.001701	1.002	7.225E-7	2.029	8.814E-10	3.031	2.696E-13	4.103

Example 5.2: Consider the following time dependent IVP

$$y'(t) = -2\pi \sin 2\pi t - 2(y - \cos 2\pi t),\ y(0) = 1,\ t \in [0, 1], \tag{5.13}$$

with the exact solution

$$y(t) = \cos(2\pi t).$$

The RIDC-FE and the RIDC-BE methods of Section 2.1 are applied to the IVP (5.13). The methods are tested for different time steps. The errors and the orders of accuracy of RIDC-FE and RIDC-BE are shown in Table 5.3 and Table 5.4 respectively.

Table 5.3: Errors and orders of accuracy of RIDC-FE method applied to the IVP (5.13). The number of subintervals in each group is $K = 20$ and the step size is $\Delta t = 0.01$.

Step size	RIDC-FE-1		RIDC-FE-2		RIDC-FE-3		RIDC-FE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.02942	—	0.0005096	—	1.083E-5	—	5.084E-7	—
$\Delta t/2$	0.01614	0.8661	0.0001315	1.954	1.545E-6	2.809	3.159E-8	4.008
$\Delta t/4$	0.008057	1.002	3.441E-5	1.934	1.959E-7	2.98	1.969E-9	4.004
$\Delta t/8$	0.004025	1.001	8.664E-6	1.99	2.468E-8	2.989	1.233E-10	3.996

Table 5.4: Errors and orders of accuracy of RIDC-BE method applied to the IVP (5.13). The number of subintervals in each group is $K = 20$ and the step size is $\Delta t = 0.01$.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.00794	—	0.0003313	—	1.9E-6	—	2.982E-7	—
$\Delta t/2$	0.003948	1.008	8.278E-5	2.001	2.226E-7	3.094	1.836E-8	4.022
$\Delta t/4$	0.001968	1.004	2.083E-5	1.991	2.619E-8	3.087	1.139E-9	4.01
$\Delta t/8$	0.0009828	1.002	5.232E-6	1.993	3.149E-9	3.056	7.095E-11	4.005

Example 5.3: Consider the following one dimensional homogeneous heat equation

$$u_t = \epsilon u_{xx},\ x \in [0,1],\ t \in [0,1.2], \tag{5.14}$$

with the initial condition

$$u(x,0) = \sin(\pi x), \tag{5.15}$$

and the boundary conditions

$$u(0,t) = u(1,t) = 0,\ t > 0, \tag{5.16}$$

where ϵ is a positive constant.

Discretizing (5.14) by central finite differences on a fixed uniform mesh with mesh size h, we obtain the following system of ODEs

$$\frac{du_j}{dt} = \frac{\epsilon}{h^2}\Big(u_{j-1} - 2u_j + u_{j+1}\Big),\ j = 2,\ldots,N-1, \tag{5.17}$$

where N is the total number of mesh points. Rewriting the boundary conditions (5.16), we have

$$\frac{du_1}{dt} = 0 \text{ and } \frac{du_N}{dt} = 0. \tag{5.18}$$

The system of ODEs (5.17) combined with boundary conditions (5.18) and the initial condition (5.15) form a system of IVPs. This system is then solved by the RIDC-FE and the RIDC-BE methods of Section 2.1. We first solve the system of ODEs (5.17) by the Matlab ODE solver $ode15s$ with a very small tolerance $tol = 10^{-14}$ and we use the solution obtained by $ode15s$ is a surrogate for the exact solution of the system of ODEs (5.17). We then compute the errors as the infinity norm of the errors at time $t = 1.2$. For the order of accuracy computation, we refer to the *Method 1* of Section 5.1.1. The errors and the orders of accuracy of the RIDC-FE and the RIDC-BE methods are recorded in Table 5.5 and Table 5.6 respectively.

Table 5.5: Errors and orders of accuracy of RIDC-FE method applied to (5.17) with $N = 10$, $\epsilon = 0.4$, $K = 5$. The step size is taken as $\Delta t = 0.005$ so that the stability criterion of forward Euler method is satisfied. Errors are computed as the infinity norm of the errors at time $t = 1.2$. The orders of accuracy are computed by *Method 1* of Section 5.1.1.

Step size	RIDC-FE-1		RIDC-FE-2		RIDC-FE-3		RIDC-FE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	1.022E-4	—	1.184E-6	—	1.95E-8	—	2.103E-10	—
$\Delta t/2$	5.186E-5	0.9791	3.069E-7	1.947	2.561E-9	2.929	1.429E-11	3.88
$\Delta t/4$	2.611E-5	0.9897	7.803E-8	1.976	3.275E-10	2.967	9.277E-13	3.945
$\Delta t/8$	1.31E-5	0.9949	1.967E-8	1.988	4.14E-11	2.984	5.903E-14	3.974
$\Delta t/16$	6.563E-6	0.9974	4.937E-9	1.994	5.203E-12	2.992	3.6E-15	4.036

Table 5.6: Errors and orders of accuracy of RIDC-BE method applied to (5.17) with $N = 10$, $\epsilon = 0.4$, $K = 5$. The step size is taken as $\Delta t = 0.01$. Errors are computed as the infinity norm of the errors at time $t = 1.2$. The orders of accuracy are computed by *Method 1* of Section 5.1.1

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.0002223	—	3.704E-6	—	1.868E-7	—	1.397E-9	—
$\Delta t/2$	0.0001082	1.039	1.09E-6	1.765	2.741E-8	2.768	1.583E-10	3.141
$\Delta t/4$	5.334E-5	1.02	2.943E-7	1.889	3.711E-9	2.885	1.249E-11	3.664
$\Delta t/8$	2.649E-5	1.01	7.641E-8	1.946	4.827E-10	2.943	8.687E-13	3.846

Example 5.4: To test our newly developed RIDC formula in Section 2.2, we consider the following IVP

$$Ly^{'}(t) = g(t, y),\ t \in [0, 1.2], \tag{5.19}$$

where $y \in \mathbb{R}^2$,

$$L = \begin{bmatrix} 4 & -1 \\ -1 & 4 \end{bmatrix},$$

and

$$g(t, y) = (y_1 + 4y_2, -4y_1 - y_2)^t.$$

The exact solution is given by $y(t) = (\sin(t), \cos(t))^T$.

The RIDC-BE method given in Section 2.2 is applied to the ODE (5.19) with initial condition $y(0) = (0, 1)^T$. The RIDC-BE method up to the order 4 is tested for different step sizes. Errors are computed as the infinity norm of the errors at the final time $t = 1.2$. The orders of accuracy of the methods are computed by *Method 1* of Section 5.1.1. The errors and the orders of accuracy of the RIDC-BE method are shown in Table 5.7.

Table 5.7: The RIDC-BE method up to the order 4 is tested with $K = 20$, $\Delta t = 0.02$. The numerical solution is compared with the exact solution, and the errors are computed as the infinity norm of the errors at time $t = 1.2$.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.005945	—	6.337E-5	—	1.398E-6	—	9.421E-9	—
$\Delta t/2$	0.002986	0.9933	1.522E-5	2.058	1.717E-7	3.025	4.744E-10	4.312
$\Delta t/4$	0.001497	0.9967	3.764E-6	2.015	2.136E-8	3.007	2.758E-11	4.105
$\Delta t/8$	0.0007492	0.9984	9.384E-7	2.004	2.667E-9	3.002	1.691E-12	4.027

Example 5.5: To verify our developed RIDC formula in Section 2.3 we choose the following IVP

$$L(y)y^{'}(t) = g(t, y),\ t \in [0, 1.2], \tag{5.20}$$

where $y \in \mathbb{R}^2$,

$$L = \begin{bmatrix} y_1^2 + 4 & -1/2 \\ -1/2 & y_2^2 + 4 \end{bmatrix},$$

and

$$g(t, y) = (y_1^2 y_2 + 4y_2 + 1/2y_1, -1/2y_2 - y_1 y_2^2 - 4y_1)^t.$$

The exact solution of the IVP (5.20) is given by $y(t) = (\sin(t), \cos(t))^T$.

The RIDC-BE method given in Section 2.3 is applied to the ODE (5.20) with initial condition $y(0) = (0, 1)^T$. The RIDC-BE method up to the order 4 is tested for different step sizes. Errors are computed as the infinity norm of the errors at the final time $t = 1.2$. The orders of accuracy of the methods are computed by *Method 1* of Section 5.1.1. The errors and the orders of accuracy of the method are shown in Table 5.8.

Table 5.8: The RIDC-BE method up to the order 4 is tested with $K = 20$, $\Delta t = 0.02$. The numerical solution is compared with the exact solution, and the errors are computed as the infinity norm of the errors at time $t = 1.2$.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.01117	—	8.315E-5	—	1.59E-6	—	8.305E-9	—
$\Delta t/2$	0.00559	0.9992	1.978E-5	2.072	1.921E-7	3.049	4.966E-10	4.064
$\Delta t/4$	0.002796	0.9996	4.806E-6	2.041	2.342E-8	3.036	2.923E-11	4.087
$\Delta t/8$	0.001398	0.9998	1.184E-6	2.022	2.886E-9	3.021	1.755E-12	4.058

5.3 A Parabolic Nonlinear PDE on a Moving Mesh

We have seen from Example 5.5 that our new RIDC-BE formula gives expected results for the small systems of ODEs $L(t, y)y' = f(t, y)$. We now consider a system of ODEs of the form $L(t, y)y^{'} = f(t, y)$ which arises from the semi-discretization of a PDE by the moving method of lines.

Example 5.6: We consider one dimensional Burgers' equation with homogeneous Dirichlet boundary conditions

$$u_t = \epsilon u_{xx} - \left(\frac{u^2}{2}\right)_x, \ x \in [0, 1], \ t > 0, \tag{5.21}$$

along with the boundary conditions

$$u(0,t) = 0 \text{ and } u(1,t) = 0, \tag{5.22}$$

and the initial condition

$$u(x,0) = \sin(2\pi x) + \frac{1}{2}\sin(\pi x), \tag{5.23}$$

where $\epsilon > 0$, is a physical parameter.

From the equations (4.45) and (4.50) of Chapter 4, we have the semi-discretized form of $1D$ Burgers' equation (5.21) on the non-uniform adaptive mesh given by

$$\frac{du_j}{dt} - \frac{(u_{j+1}-u_{j-1})}{(x_{j+1}-x_{j-1})}\frac{dx_j}{dt} = \frac{2\epsilon}{(x_{j+1}-x_{j-1})}\Big[\frac{u_{j+1}-u_j}{x_{j+1}-x_j} - \frac{u_j-u_{j-1}}{x_j-x_{j-1}}\Big] - \frac{1}{2}\frac{(u_{j+1}^2-u_{j-1}^2)}{(x_{j+1}-x_{j-1})},\ j = 2,\ldots,N-1, \tag{5.24}$$

along with the discretized form of mesh equation (4.50) given by

$$\frac{dx_{j-1}}{dt} - 2\frac{dx_j}{dt} + \frac{dx_{j+1}}{dt} = -\frac{1}{\tau}\Big[\frac{\rho_{j+1}+\rho_j}{2}(x_{j+1}-x_j) - \frac{\rho_j+\rho_{j-1}}{2}(x_j-x_{j-1})\Big],\ j = 2,\ldots,N-1, \tag{5.25}$$

where x_j and u_j are the mesh points and the solutions for the solution of (5.21). Here $\epsilon > 0$ and $\tau > 0$ are positive constants, and ρ is the mesh density function.

The system of ODEs (5.24) and (5.25) along with the boundary conditions

$$\frac{du_1}{dt} = 0,\ \frac{du_N}{dt} = 0, \tag{5.26}$$

and

$$\frac{dx_1}{dt} = 0,\ \frac{dx_N}{dt} = 0, \tag{5.27}$$

form a coupled system of IVPs of the form

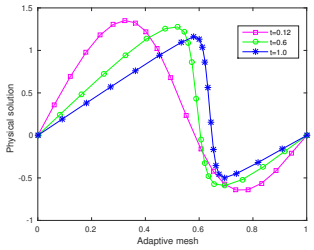
$$L(t,y)y' = f(t,y), \tag{5.28}$$

where $y = [u_1,\ldots,u_N,x_1,\ldots,x_N]$ and $y' = \big[u_1',\ldots,u_N',x_1',\ldots,x_N'\big]$. Here $L(t,y)$ is

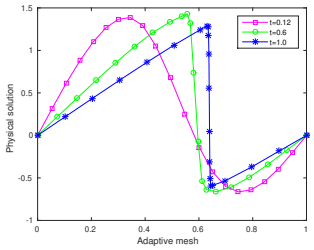
a $2N \times 2N$ matrix given in Section 4.2.2, and $f(t, y)$ is the right side of the coupled system of ODEs obtained by combining (5.24) and (5.25).

We begin by using the curvature based mesh density function given in Section 4.2.2 and we apply the RIDC-BE method of Section 2.3 to the system of IVPs (5.28). We choose $\tau = \frac{1}{10}$ and the method is tested for different values the parameters N and ϵ. The computed solutions at time $t = 0.12$, 0.6, and 1.0 are shown in Fig. 5.1.

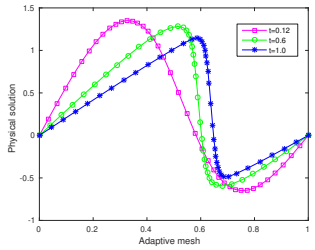
To verify the order of accuracy of the solution, we solve the system (5.28) by Matlab ODE solver *ode*15*s* with a very small tolerance $tol = 10^{-14}$ and consider that solution as the exact solution of (5.28). We then approximate the orders of accuracy of the solution by the *Method 1* of Section 5.1.1. The errors are computed as the infinity norm of the errors at time $t = 0.12$. The errors and the orders of accuracy of the RIDC-BE method for different values of N and ϵ are recorded in Table 5.9, Table 5.10, and Table 5.11.



(a) Solution at $t = 0.12$, 0.6, and 1.0 for $N = 21$, $\epsilon = 0.01$.



(b) Solution at $t = 0.12$, 0.6, and 1.0 for $N = 21$, $\epsilon = 0.001$.



(c) Solution at $t = 0.12$, 0.6, and 1.0 for $N = 41$, $\epsilon = 0.01$.

Fig. 5.1: Solution of Burgers' equation (5.21) by adaptive RIDC-BE method at times $t = 0.12$, 0.6, and 1.0 with (a) $N = 21$, $\epsilon = 0.01$, $\Delta t = 0.01$, (b) $N = 21$, $\epsilon = 0.001$, $\Delta t = 0.01$ and (c) $N = 41$, $\epsilon = 0.01$, $\Delta t = 0.01$.

Table 5.9: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 21$, $\Delta t = 0.01$, $\epsilon = 10^{-2}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 1* of Section 5.1.1

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.01458	—	0.01215	—	0.003346	—	0.005664	—
$\Delta t/2$	0.008398	0.7958	0.002837	2.099	0.0002653	3.657	0.000814	2.799
$\Delta t/4$	0.004617	0.8631	0.000533	2.412	1.445E-5	4.199	7.517E-5	3.437
$\Delta t/8$	0.002442	0.9189	8.413E-5	2.664	9.659E-6	0.581	5.146E-6	3.869
$\Delta t/16$	0.001259	0.9556	1.198E-5	2.812	2.157E-6	2.163	3.037E-7	4.083
$\Delta t/32$	0.0006398	0.9767	2.0E-6	2.583	3.57E-7	2.595	1.712E-8	4.149

Table 5.10: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 41$, $\Delta t = 0.01$, $\epsilon = 10^{-2}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 1* of Section 5.1.1

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.02305	—	0.01093	—	0.002663	—	0.005308	—
$\Delta t/2$	0.01342	0.7801	0.002414	2.178	0.0001713	3.958	0.0007396	2.843
$\Delta t/4$	0.007493	0.841	0.0004282	2.495	2.248E-5	2.93	6.65E-5	3.475
$\Delta t/8$	0.003956	0.9216	6.052E-5	2.823	8.741E-6	1.363	5.272E-6	3.657
$\Delta t/16$	0.002022	0.9683	1.102E-5	2.457	1.932E-6	2.178	3.001E-7	4.135
$\Delta t/32$	0.001023	0.983	2.361E-6	2.223	3.238E-7	2.576	9.545E-9	4.974

We notice from Table 5.9-5.11 that the order of accuracy of the 3^{rd} order RIDC method using the curvature mesh density function is slightly different from our expectation.

To test this further we experiment with another mesh density function. We repeat the computation using arclength mesh density function given in Section 4.2.2 with smoothing parameters $\gamma = 2$ and $p = 2$. The errors and the orders of accuracy of the method for different values of t, Δt and ϵ are recorded in Table 5.12 - 5.17. The orders of accuracy of the solution are computed by the *Method 2* of Section 5.1.1.

Table 5.11: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 21$, $\Delta t = 0.01$, $\epsilon = 10^{-3}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 1* of Section 5.1.1

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	0.02754	—	0.01865	—	0.004729	—	0.007633	—
$\Delta t/2$	0.01587	0.7947	0.004398	2.084	0.0003511	3.751	0.001034	2.883
$\Delta t/4$	0.008516	0.8985	0.0008543	2.364	2.48E-5	3.824	8.983E-5	3.525
$\Delta t/8$	0.004443	0.9385	0.0001445	2.563	1.349E-5	0.8787	5.957E-6	3.915
$\Delta t/16$	0.002274	0.9663	2.368E-5	2.61	2.845E-6	2.245	3.499E-7	4.089
$\Delta t/32$	0.001151	0.9823	4.186E-6	2.5	4.572E-7	2.637	1.997E-8	4.131

Table 5.12: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 21$, $\Delta t = 0.0012$, $\epsilon = 10^{-2}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 2* of Section 5.1.1.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	—	—	—	—	—	—	—	—
$\Delta t/2$	0.0017	–	6.695E-5	–	2.661E-6	–	1.610E-7	–
$\Delta t/4$	0.00085	0.967	1.686E-5	1.989	3.576E-7	2.895	9.746E-9	4.046
$\Delta t/8$	0.00043	0.984	3.264E-6	2.369	4.811E-8	2.894	5.823E-10	4.065
$\Delta t/16$	0.00022	0.992	5.848E-7	2.481	3.746E-9	2.834	3.856E-11	3.917

5.4 Runtime Comparison and Discussion of Results

In examples 5.1, 5.2, 5.4 and 5.5, RIDC methods are applied to the given IVPs. Both the explicit and the implicit RIDC methods give the order of accuracy as expected from the theoretical predictions. For example, a 3^{rd} order RIDC method is supposed to give a 3^{rd} order accurate solution and a 4^{th} order RIDC method is supposed to give a 4^{th} order accurate solution. For the examples 5.3 and 5.6, the physical PDEs are discretized first by the method of lines and the moving method of lines respectively to obtain systems of IVPs. RIDC methods are then applied to the resulting systems

Table 5.13: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 21$, $\Delta t = 0.0012$, $\epsilon = 10^{-3}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 2* of Section 5.1.1.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	—	—	—	—	—	—	—	—
$\Delta t/2$	0.0018	–	8.060E-5	–	3.082E-6	–	2.098E-7	–
$\Delta t/4$	0.00094	0.972	2.008E-5	2.004	4.008E-7	2.942	1.268E-8	4.048
$\Delta t/8$	0.00047	0.986	3.786E-6	2.408	5.247E-8	2.933	7.544E-9	4.071
$\Delta t/16$	0.00024	0.993	6.742E-7	2.489	4.647E-9	2.837	3.856E-11	4.021

Table 5.14: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 21$, $\Delta t = 0.0012$, $\epsilon = 10^{-4}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 2* of Section 5.1.1.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	—	—	—	—	—	—	—	—
$\Delta t/2$	0.0019	–	8.487E-5	–	3.212E-6	–	2.172E-7	–
$\Delta t/4$	0.00095	0.972	2.097E-5	2.017	4.089E-7	2.973	1.311E-8	4.051
$\Delta t/8$	0.00048	0.986	3.942E-6	2.411	5.335E-8	2.938	7.804E-10	4.070
$\Delta t/16$	0.00024	0.993	6.864E-7	2.522	4.810E-9	2.840	4.827E-11	4.015

Table 5.15: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 0.12]$ with $N = 21$, $\Delta t = 0.0012$, $\epsilon = 10^{-5}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 0.12$ and the orders of accuracy are approximated by the *Method 2* of Section 5.1.1.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	—	—	—	—	—	—	—	—
$\Delta t/2$	0.0019	–	8.532E-5	–	3.226E-6	–	2.165E-7	–
$\Delta t/4$	0.00094	0.972	2.106E-5	2.017	4.097E-7	2.977	1.307E-8	4.051
$\Delta t/8$	0.00048	0.986	3.959E-6	2.411	5.343E-8	2.939	7.780E-10	4.070
$\Delta t/16$	0.00024	0.993	6.876E-7	2.525	7.460E-9	2.840	3.856E-11	4.016

Table 5.16: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 1]$ with $N = 21$, $\Delta t = 0.001$, $\epsilon = 10^{-2}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 1$ and the orders of accuracy are approximated by the *Method 2* of Section 5.1.1.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	—	—	—	—	—	—	—	—
$\Delta t/2$	4.119E-4	–	6.606E-6	–	5.058E-7	–	4.910E-8	–
$\Delta t/4$	2.048E-4	1.008	1.477E-6	2.161	7.083E-8	2.836	3.090E-9	3.990
$\Delta t/8$	1.021E-4	1.004	2.110E-7	2.807	9.572E-9	2.887	1.952E-10	3.984
$\Delta t/16$	5.097E-5	1.002	5.848E-7	2.114	1.347E-9	2.828	1.265E-11	3.948

Table 5.17: The errors and the orders of accuracy of the RIDC-BE method applied to the coupled IVP system (5.28) in the time interval $[0, 1]$ with $N = 21$, $\Delta t = 0.001$, $\epsilon = 10^{-3}$ and $\tau = \frac{1}{10}$. The errors are computed as the infinity norm of errors at time $t = 1$ and the orders of accuracy are approximated by the *Method 2* of Section 5.1.1.

Step size	RIDC-BE-1		RIDC-BE-2		RIDC-BE-3		RIDC-BE-4	
	Errors	Orders	Errors	Orders	Errors	Orders	Errors	Orders
Δt	—	—	—	—	—	—	—	—
$\Delta t/2$	4.360E-4	–	9.198E-6	–	4.244E-7	–	6.160E-8	–
$\Delta t/4$	2.178E-4	1.001	1.848E-6	2.315	6.408E-8	2.727	3.892E-9	3.984
$\Delta t/8$	1.088E-4	1.001	4.122E-7	2.164	8.824E-9	2.861	2.464E-10	3.981
$\Delta t/16$	5.439E-5	1.000	9.725E-8	2.084	1.530E-9	2.527	1.660E-11	3.892

of IVPs. The orders of accuracy of the RIDC methods for these examples are very close to the expected theoretical results, and the orders of accuracy get closer to the desired result as the time step size gets smaller. For instance, from the Table 5.5 we notice that the order of accuracy of a 4^{th} order RIDC method gets closer to 4 as the step size Δt gets smaller. For the adaptive mesh example there is a slight departure from the expected order of accuracy of the RIDC method. There are several possible explanations. We do not have the exact solution for Burgers' equation, and we consider the approximate solution computed by Matlab $ode15s$ with very small tolerance ($tol = 10^{-14}$) as the exact solution. This might affect the computed order of accuracy of the method. We note that when we repeat the computation using the smoothed arclength monitor function and calculate the order of accuracy of the method using approximate solutions (*Method 2* of Section 5.1.1) the order of convergence meets our expectation.

In order to check the running time of RIDC method, we compute the average running time of RIDC-BE method of Section 2.3 applied to the Burgers' equation (Example 5.6). Table 5.18 gives the results for the Burgers equation (5.21) on a fixed mesh and Table 5.19 for the Burgers equation (5.21) on moving mesh using the curvature based mesh density function.

In Table 5.18, RIDC-BE-2, RIDC-BE-3 and RIDC-BE-4 are computed by two, three and four computing cores respectively and RIDC-BE-1 is the backward Euler method computed by a single computing core. We choose $n = 500$ time steps, the number of subintervals in each group is chosen to be $K = 500$, and the number of spatial mesh points is $N = 21$ and $\epsilon = 10^{-2}$. The actual time is the time taken by each method to obtain the solution at the final time $t = 1$. The ratios (μ) of the time taken by RIDC-BE-2, RIDC-BE-3, RIDC-BE-4 methods to the time taken by a single core backward Euler method are shown in Table 5.18. The theoretical time and the theoretical values of μ are calculated based on the theory discussed in Section 2.4.4.

Table 5.18: Theoretical and actual running time and the errors of the RIDC-BE method applied to the Burgers' equation (5.21) on a fixed mesh. Here we choose $N = 21$ spatial mesh points, $n = 500$ time points, $K = 500$ subintervals in each group, and $\epsilon = 10^{-2}$.

Method	Error	Theoretical Time	Actual Time	Theoretical μ	Actual μ
RIDC-BE-1	4.18E-3	–	2.781s	–	–
RIDC-BE-2	1.84E-3	2.783s	3.032s	1.001	1.090
RIDC-BE-3	5.90E-6	2.789 s	3.149s	1.003	1.132
RIDC-BE-4	3.70E-8	2.798s	3.254s	1.006	1.170

Table 5.19: Theoretical and actual running time and and the errors of the RIDC-BE method applied to the Burgers' equation (5.21) on a moving mesh. We choose $N = 21$ spatial mesh points, $n = 500$ time points, $K = 500$ subintervals in each group, and $\epsilon = 10^{-2}$.

Method	Error	Theoretical Time	Actual Time	Theoretical μ	Actual μ
RIDC-BE-1	8.319E-4	–	21.155s	–	–
RIDC-BE-2	2.58E-5	21.176s	21.501s	1.001	1.0164
RIDC-BE-3	2.98E-6	21.218s	22.241s	1.003	1.051
RIDC-BE-4	7.580E-7	21.282s	23.928s	1.006	1.131

The above Tables 5.18 and 5.19 tell us that the actual values of μ are slightly larger than the theoretical prediction. A good explanation of this issue is that the

actual computation experiences inter-communication and latency, whereas the theoretical prediction does not include this additional cost [13]. The actual μ values obtained here are similar to those reported in [13]. The reader will notice that the actual cpu time is much larger for the moving mesh simulations than for the fixed mesh case. It is important to notice that the error in the moving mesh case is less than the error in the fixed mesh case except for the RIDC-BE-4 results. This suggests that a fairer comparison of efficiency would require a larger number of fixed mesh points which would increase the required cpu time. Also, further tuning of the moving mesh code is most certainly possible. The purpose here was not to compare fixed mesh simulations versus moving mesh simulations but use these examples to show that the RIDC implementations work and scale as expected as the order increases.

Chapter 6

Conclusion and Possible Future Work

In this thesis, our ultimate goal was to apply the time parallel RIDC method to the system of IVPs which arises from the semi-discretization of PDEs by the moving method of lines. We show the necessary derivation, discussion and relevant examples dividing them into several chapters.

In Chapter 1, we give a very introductory review of the numerical methods for the solution of initial value problems, for ordinary differential equations and partial differential equations. We briefly discuss the most commonly used sequential time stepping methods, deferred correction methods, and different types of parallel methods for ODEs. We also discuss the need for adaptive mesh methods for the numerical solutions of partial differential equations.

In Chapter 2, we show the derivation of the RIDC method for three different types of initial value problems : (i) $y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ (ii) $Ly^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ where, L is a constant $n \times n$ matrix, and (iii) $L(t, y)y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ where, $L(t, y)$ is square matrix. In each of the three cases the formulation of the error equations are shown, the choice of the integrator and the quadrature rules are properly addressed, and a step by step procedure of the method is presented in an algorithmic approach. We also illustrate the multi-core implementation of the predictor and the corrector.

In Chapter 3, we review the library from [40] which provides a time parallel

solution to an initial value problem of the form $y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ using RIDC. We show how the computation of the correction formulas in the explicit and implicit RIDC library are achieved. The main result of Chapter 3 is the demonstration of how the RIDC library can be used to implement the formulas constructed in Chapter 2 for IVPs of the forms (ii) and (iii) given in the previous paragraph.

In Chapter 4, the adaptive mesh generation technique using the equidistribution principle is discussed with an appropriate example. We briefly describe the equidistribution principle, the choice of mesh density functions or monitor functions, and the moving mesh partial differential equations (MMPDEs). Finally, we discretize the physical PDE and moving mesh PDE by the moving method of lines which gives a coupled system of ODEs. We then solve the coupled system of ODEs using the RIDC method. Here, backward Euler is used as predictor and corrector.

In Chapter 5, we illustrate the efficiency and the accuracy of the developed RIDC methods using several examples. For each of those examples, we investigate the errors and the order of accuracy of the method and record the results in the corresponding tables. The first three examples (Example 5.1-5.3) apply the RIDC method to solve the initial value problem of the form $y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$. Example 5.4 and Example 5.5 are associated with the RIDC methods applied to the initial value problems of types $Ly^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ where, L is a constant square matrix, and $L(t, y)y^{'}(t) = f(t, y)$, $y \in \mathbb{R}^n$ where, $L(t, y)$ is a square matrix respectively. The last example (Example 5.6) concerns applying the RIDC method to an adaptive mesh example. We choose Burgers' equation as an example of a nonlinear parabolic partial differential equation. We verify the order of accuracy of the method for different values of ϵ and different number of mesh points N.

In this thesis, we constructed a RIDC method with a spatial adaptive mesh using backward Euler as predictor and corrector. In the future, we want to test the spatial adaptive RIDC method using adaptive time stepping [7]. Adaptive time stepping is usually the way the moving method of lines is implemented. We also wish to explore (adaptive) higher order methods in space to match the high order methods in time provided by RIDC. We may wish to provide an adaptive time parallel approach using RIDC and moving meshes in two and three spatial dimensions.

Bibliography

[1] U. M. Ascher and L. R. Petzold. *Computer methods for ordinary differential equations and differential-algebraic equations*, volume 61. SIAM, 1998.

[2] W. Auzinger, H. Hofstätter, W. Kreuzer, and E. Weinmüller. Modified defect correction algorithms for ODEs. part i: General theory. *Numerical Algorithms*, 36(2):135–155, 2004.

[3] A. Bellen and M. Zennaro. Parallel algorithms for initial-value problems for difference and differential equations. *Journal of Computational and Applied Mathematics*, 25(3):341–350, 1989.

[4] K. Burrage. Parallel methods for initial value problems. *Applied Numerical Mathematics*, 11(1-3):5–25, 1993.

[5] K. Burrage. Parallel methods for ODEs. *Advances in Computational Mathematics*, 7(1-2):1–31, 1997.

[6] P. Chartier and B. Philippe. A parallel shooting technique for solving dissipative ODE's. *Computing*, 51(3-4):209–236, 1993.

[7] A. Christlieb, C. Macdonald, B. Ong, and R. Spiteri. Revisionist integral deferred correction with adaptive step-size control. *Communications in Applied Mathematics and Computational Science*, 10(1):1–25, 2015.

[8] A. Christlieb, M. Morton, B. Ong, and J. Qiu. Semi-implicit integral deferred correction constructed with additive Runge–Kutta methods. *Communications in Mathematical Sciences*, 9(3):879–902, 2011.

[9] A. Christlieb and B. Ong. Implicit parallel time integrators. *Journal of Scientific Computing*, 49(2):167–179, 2011.

[10] A. Christlieb, B. Ong, and J. Qiu. Comments on high-order integrators embedded within integral deferred correction methods. *Communications in Applied Mathematics and Computational Science*, 4(1):27–56, 2009.

[11] A. Christlieb, B. Ong, and J. Qiu. Integral deferred correction methods constructed with high order Runge-Kutta integrators. *Mathematics of Computation*, 79(270):761–783, 2010.

[12] A. J. Christlieb, R. D. Haynes, and B. W. Ong. A parallel space-time algorithm. *SIAM Journal on Scientific Computing*, 34(5):C233–C248, 2012.

[13] A. J. Christlieb, C. B. Macdonald, and B. W. Ong. Parallel high-order integrators. *SIAM Journal on Scientific Computing*, 32(2):818–835, 2010.

[14] A. Dutt, L. Greengard, and V. Rokhlin. Spectral deferred correction methods for ordinary differential equations. *BIT Numerical Mathematics*, 40(2):241–266, 2000.

[15] M. Emmett, M. L. Minion, et al. Toward an efficient parallel in time method for partial differential equations. *Communications in Applied Mathematics and Computational Science*, 7(1):105–132, 2012.

[16] R. Falgout, T. Manteuffel, B. Southworth, and J. Schroder. Parallel-in-time for moving meshes. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2016.

[17] R. D. Falgout, S. Friedhoff, T. V. Kolev, S. P. MacLachlan, and J. B. Schroder. Parallel time integration with multigrid. *SIAM Journal on Scientific Computing*, 36(6):C635–C661, 2014.

[18] R. D. Falgout, T. A. Manteuffel, B. O'Neill, and J. B. Schroder. Multigrid reduction in time for nonlinear parabolic problems: A case study. *SIAM Journal on Scientific Computing*, 39(5):S298–S322, 2017.

[19] M. J. Gander. Overlapping Schwarz for linear and nonlinear parabolic problems. *In Proceedings of the 9th International Conference on Domain Decomposition, pp. 97–104. ddm.org*, 1996.

[20] M. J. Gander. 50 years of time parallel time integration. In *Multiple Shooting and Time Domain Decomposition Methods*, pages 69–113. Springer, 2015.

[21] M. J. Gander, L. Halpern, and F. Nataf. Optimal convergence for overlapping and non-overlapping Schwarz waveform relaxation. *In Proceedings of the 11th International Conference of Domain Decomposition Methods, C.-H. Lai, P. Bjørstad, M. Cross, and O. Widlund,eds.*, 1999.

[22] M. J. Gander and M. Neumuller. Analysis of a new space-time parallel multigrid algorithm for parabolic problems. *SIAM Journal on Scientific Computing*, 38(4):A2173–A2208, 2016.

[23] M. J. Gander and S. Vandewalle. On the superlinear and linear convergence of the parareal algorithm. In *Domain decomposition methods in science and engineering XVI*, pages 291–298. Springer, 2007.

[24] T. Hagstrom and R. Zhou. On the spectral deferred correction of splitting methods for initial value problems. *Communications in Applied Mathematics and Computational Science*, 1(1):169–205, 2007.

[25] G. Horton and S. Vandewalle. A space-time multigrid method for parabolic partial differential equations. *SIAM Journal on Scientific Computing*, 16(4):848–864, 1995.

[26] W. Huang, Y. Ren, and R. D. Russell. Moving mesh methods based on moving mesh partial differential equations. *Journal of Computational Physics*, 113(2):279–290, 1994.

[27] W. Huang, Y. Ren, and R. D. Russell. Moving mesh partial differential equations (MMPDES) based on the equidistribution principle. *SIAM Journal on Numerical Analysis*, 31(3):709–730, 1994.

[28] W. Huang and R. D. Russell. *Adaptive moving mesh methods*, volume 174. Springer Science & Business Media, 2010.

[29] K. R. Jackson. A survey of parallel numerical methods for initial value problems for ordinary differential equations. *IEEE Transactions on Magnetics*, 27(5):3792–3797, 1991.

[30] D. Ketcheson and U. bin Waheed. A comparison of high-order explicit runge–kutta, extrapolation, and deferred correction methods in serial and parallel. *Communications in Applied Mathematics and Computational Science*, 9(2):175–200, 2014.

[31] A. Layton and M. Minion. Implications of the choice of predictors for semi-implicit Picard integral deferred correction methods. *Communications in Applied Mathematics and Computational Science*, 2(1):1–34, 2007.

[32] A. T. Layton and M. L. Minion. Conservative multi-implicit spectral deferred correction methods for reacting gas dynamics. *Journal of Computational Physics*, 194(2):697–715, 2004.

[33] A. T. Layton and M. L. Minion. Implications of the choice of quadrature nodes for Picard integral deferred corrections methods for ordinary differential equations. *BIT Numerical Mathematics*, 45(2):341–373, 2005.

[34] R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, volume 98. SIAM, 2007.

[35] Y. Liu, C. Shu, and M. Zhang. Strong stability preserving property of the deferred correction time discretization. *Journal of Computational Mathematics*, 26:633–656, 2008.

[36] M. L. Minion. Semi-implicit spectral deferred correction methods for ordinary differential equations. *Communications in Mathematical Sciences*, 1(3):471–500, 2003.

[37] W. Miranker. A survey of parallelism in numerical analysis. *Siam Review*, 13(4):524–547, 1971.

[38] J. Nievergelt. Parallel methods for integrating ordinary differential equations. *Communications of the ACM*, 7(12):731–733, 1964.

[39] B. Ong, A. Melfi, and A. Christlieb. Parallel semi-implicit time integrators. *arXiv preprint arXiv:1209.4297*, 2012.

[40] B. W. Ong, R. D. Haynes, and K. Ladd. Algorithm 965: RIDC methods: A family of parallel time integrators. *ACM Transactions on Mathematical Software (TOMS)*, 43(1):8, 2016.

[41] J. M. Stockie, J. A. Mackenzie, and R. D. Russell. A moving mesh method for one-dimensional hyperbolic conservation laws. *SIAM Journal on Scientific Computing*, 22(5):1791–1813, 2001.

[42] S. Vandewalle. *Parallel multigrid waveform relaxation for parabolic problems*. B.G. Teubner, Stuttgart, 1993.

[43] S. Vandewalle and D. Roose. The parallel waveform relaxation multigrid method. *In Parallel Processing for Scientific Computing, G. Rodrigue, ed.,Society for Industrial and Applied Mathematics Philadelphia, PA*, pages 152–156, 1989.

Appendix A

Computer Codes

A.1 Matlab Code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% RIDC-FE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function Name : ridc_fe                                                 %
% Description : RIDC method method using forward Euler%
% for ODE of the form y' = f(t,y) with appropriate IC %
% Inputs:                                                                 %
% f = right hand side function                                            %
% p = order of the method                                                 %
% y0 = initial condition                                                  %
% tspan = time interval                                                   %
% dt = step size                                                          %
% K = number of subintervals in group                                     %
% Output : p^th order solution                                            %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function rh=ridc_fe(f,p,y0,tspan,dt,K)
    N=length(y0);
    M=p-1;                    % number of corrections required.
    T=tspan(1,2);             % final time
    W=int16(T/dt);            % total time intervals
```

```
J=int16(W/K);              % number of groups
% empty vectors
y=zeros(N,J+1);
u=zeros(N,K+1,M+1);
y(:,1)=y0;                % initial condition
s=integration_matrix(p1);% integration matrix
t=zeros(K+1,1);
J=double(J);
t0=tspan(1);              % starting time
for j=1:J
    u(:,1,1)=y(:,j);      % copying initial solution to each group
    % prediction loop
    t(1)=t0;              % time initialization  in each group
    for m=1:K
        t(m+1)=t(1)+m*dt;
        u(:,m+1,1)= u(:,m,1) + dt*f(t(m),u(:,m,1));
    end
% correction loop
   for l=1:M
       u(:,1,l+1)=u(:,1,l);
       for m=1:l
           s1=0;
           for i=1:l+1
               s1=s1+s(m,i,l)*f(t(1)+(i-1)*dt,u(:,i,l));     % residual part
           end
           u(:,m+1,l+1)=u(:,m,l+1)+dt*( f(t(m),u(:,m,l+1))-f( t(m),u(:,m,l)))+
           l*dt*s1;
       end
       for m=l+1:K
           s2=0;
           for i=1:l+1
             s2=s2+s(l,i,l)*f(t(m-l+i),u(:,m-l+i,l));
           end
        u(:,m+1,l+1)=u(:,m,l+1)+dt*( f(t(m),u(:,m,l+1))-f( t(m),u(:,m,l) ) ) +
```

```
            l*dt*s2;
           end
      end
    y(:,j+1)=u(:,K+1,M+1);    % updating solution for the next group
    t0=t(K+1,1);              % updating time for the next group
  end
 rh=y;
 end
end
%%%%%%%%%%%%%%%%%%%%%%%%  End ridc_fe %%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%% RIDC-BE %%%%%%%%%%%%%%%%%%%%%%%%
% Function Name : ridc_be                                %
% Description: RIDC method method using backward Euler%
% for ODE of the form y' = f(t,y) with appropriate IC %
% Inputs:                                                %
% f = right hand side function                           %
% p = order of the method                                %
% y0 = initial condition                                 %
% tspan = time interval                                  %
% dt = step size                                         %
% K = number of subintervals in group                    %
% Output : p^th order solution                           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function rh=ridc_be(f,p,u0,tspan,dt,K)
   N=length(u0);
   M=p-1;                  % number of corrections required.
   T=tspan(1,2);           % final time
   W=int16(T/dt);          % total time intervals
   J=int16(W/K);           % number of groups
   y=zeros(N,J+1);
   u=zeros(N,K+1,M+1);
   y(:,1)=u0;              % initial condition
```

```
t=zeros(K+1,1);
J=double(J);
M=double(M);
t0=tspan(1);              % starting time
s=integration_matrix(p); % integration matrix
max_iter=100;             % max iteration for Newton's method
tol=1e-8;                 % stopping tolerance for Newton's method

for j=1:J
    u(:,1,1)=y(:,j);      % copying initial solution to each group
    t(1)=t0;              % time initialization  in each group
  % prediction loop
    for m=1:K
        t(m+1)=((j-1)*K+(m-1))*dt;
        v=u(:,m,1);
  % start newton loop
        for p=1:max_iter
            Fn=fun_pred(v,u(:,m,1),t(m+1),dt,f);
            % numerical Jacobian
            Jn=jacobFD(@fun_pred,v,u(:,m,1),t(m+1),dt,f);
            z=v-Jn\Fn;
               if norm(abs(z-v),inf)<tol
                  break
               end
            v=z;
        end
 % end newton loop
       u(:,m+1,1) = z;
   end
   % correction loop
   for l=1:M
       u(:,1,l+1)=u(:,1,l); % initial guess
       for m=1:l
           v=u(:,m,l+1);
```

```
      q=l;
      s1=0;
      % residual part
      for n1=1:l+1
          s1=s1+s(m,n1,l)*f(t(1)+(n1-1)*dt,u(:,n1,l));
      end
% newton loop
     for p=1:max_iter
         Fn=fun_corr(v,u(:,:,:),t(m+1),dt,s1,m,q,f,l);
         % numerical Jacobian
         Jn=jacobFD(@fun_corr,v,u(:,:,:),t(m+1),dt,s1,m,q,f,l);
         z=v-Jn\Fn;
           if norm(abs(z-v),inf)<tol
              break
           end
         v=z;
     end
     u(:,i+1,l+1) = z;
  end

 for i=l+1:K
     v=u(:,i,l+1);
     m=i;
     q=l;
     s2=0;
     % residual part
     for n1=1:l+1
         s2=s2+s(l,n1,l)*f(t(m-l+n1),u(:,m-l+n1,l));
     end
    % newton loop starts
        for p=1:max_iter
            Fn=fun_corr(v,u(:,:,:),t(m+1),dt,s2,m,q,f,l);
            % numerical Jacobian
            Jn=jacobFD(@fun_corr,v,u(:,:,:),t(m+1),dt,s2,m,q,f,l);
```

```
                    z=v-Jn\Fn;
                    if norm(abs(z-v),inf)<tol
                      break
                    end
                    v=z;
                end
                % end newton loop
            u(:,i+1,l+1) = z;
        end
      end
    y(:,j+1)= u(:,K+1,M+1);
    t0=t(K+1,1);
  end
 rh=y;
 end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  End ridc_be %%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Integration Matrix %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function Name : integration_matrix                                      %
% Description: the function integration_matrix takes an input m    %
% gives an integration matrix which contains the quadrature weights%
% Inputs:                                                                 %
% m = an integer greater than 1                                           %
% Output : a matrix which contains the quadrature weights          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function rh = integration_matrix(m)
    format compact, format long
    syms x;
    p=m;
    M=p-1;
    S=zeros(p-1,p,p-1);
```

```
    l=1;
    while(l<=M)
         u=linspace(0,1,l+1);
         ln=length(u);
         m=1;
         while(m<=l)
            for i=1:ln
            y=1;
            denom=1;
            integrant=1;
            for k=1:ln
              if i~=k
                y=y*(x-u(k));
               denom=denom*(u(i)-u(k));
              end
            end
        integrant=integrant*y;
        S(m,i,l)=int (integrant,u(m),u(m+1))/denom;
       end
     m=m+1;
   end
  l=l+1;
 end
 rh=S;
 end
%%%%%%%%%%%%%%%%%%%% End integration_matrix %%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%% Numerical Jacobian %%%%%%%%%%%%%%%%%%%%
% Function Name : jacobFD                                              %
% Description: the function jacobFD takes an arbitrary function g  %
% and computes the numerical Jacobian by finite difference method  %
% Inputs:                                                              %
% g = an arbitrary function                                            %
```

```
% x = initial guess                                                    %
% Output : Jacobian matrix J                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function J = jacobFD(g,x,varargin)
    delx=1e-8;
    m=length(x);
    J=zeros(m,m);
    for j = 1:m
        xx = x;
        xx(j) = x(j) + delx;
        f1=feval(g,x,varargin{:});
        f2=feval(g,xx,varargin{:});
        J(:,j) = (f2-f1)/delx;
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%% End jacobFD %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Predictor %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function Name : fun_pred                                            %
% Description: the function fun_pred evaluates the function value     %
% Inputs:                                                             %
% p = unknown parameter                                               %
% u_old = known value from step                                       %
% t = current time                                                    %
% dt = step size                                                      %
% f = current function                                                %
% Output : Jacobian matrix J                                          %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function F = fun_pred(p,u_old,t,dt,f)
     F= p-dt*f(p)-u_old;
end
%%%%%%%%%%%%%%%%%%%%%%%%%% fun_pred %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%% Corrector %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function Name : fun_corr                                                    %
% Description: the function fun_corr evaluates correction formula  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function F = fun_corr(p,u,t,dt,s1,m,l,f,M)
     F= p-dt*( f(t,p) -f(t,u(:,m+1,l)) ) - u(:,m,l+1) - M*dt*s1;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% fun_corr %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%% Right hand side functions %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%% for example 5.1 %%%%%%%%%%%%%%%%%%%%%%%

function rh=f1(t,x)
    rh= x;
end

%%%%%%%%%%%%%%%%%%%% for example 5.2 %%%%%%%%%%%%%%%%%%%%%%%

function rh=f2(t,x)
    rh=-2*pi*sin(2*pi*t)-2*( x-cos(2*pi*t) );
end

%%%%%%%%%%%%%%%%%%%% for example 5.3 %%%%%%%%%%%%%%%%%%%%%%%

function F=f3(t,u)
    ep=4e-1;
    N=length(u);
    h=1/(N);
```

```
    F=zeros(N,1);
    F(1)=0; % boundary condition
    for j=2:N-1
        F(j)= (ep/h.^2)*( u(j-1) -2*u(j) + u(j+1) );
    end
    F(N)=0; % boundary condition
end

%%%%%%%%%%%%%%%%%%% for example 5.4 %%%%%%%%%%%%%%%%%%%%%

function rh=f4(t,x)
% Mass matrix
    L=[4 -1;
      -1 4];
    f=[x(1)+4*x(2);
      -4*x(1)-x(2)];
% linear solve
    h=L\f;
    rh= h;
end

%%%%%%%%%%%%%%%%%%% for example 5.5 %%%%%%%%%%%%%%%%%%%%%

function rh=f5(t,x)

    % Mass matrix
    L=[x(1)^2+4 -1/2;
       -1/2 x(2)^2+4];
    f=[x(1)^2*x(2)+4*x(2)+(1/2)*x(1);
       -(1/2)*x(2)-x(1)*x(2)^2-4*x(1)];
% linear solve
    h=L\f;
    rh= h;
end
```

```
%%%%%%%%%%%%%%%%%%% for example 5.6 %%%%%%%%%%%%%%%%%%%%

function rh = f6(y)
     neq=length(y);
     N=neq/2;
     ep=1e-2;
     tau=1e-2;
     u = y(1:N);
     x = y(N+1:end);
     x0 = 0.0;
     u0 = 0.0;
     xNP1 = 1.0;
     uNP1 = 0.0;
     g = zeros(neq,1);

     for i = 2:N-1
         dx = x(i+1) - x(i-1);
         g(i) = (2*ep)/dx*((u(i+1)-u(i))/(x(i+1)-x(i))-(u(i)-u(i-1))/(x(i)-...
         x(i-1)) )- 0.5*(u(i+1)^2 - u(i-1)^2)/dx;
     end
     dx = x(2) - x0;
     g(1) = (2*ep)/dx*((u(2) - u(1))/(x(2) - x(1)) - (u(1) - u0)/(x(1) - x0)) -...
        0.5*(u(2)^2 - u0^2)/dx;
     dx = xNP1 - x(N-1);
     g(N) = (2*ep)/dx*((uNP1 - u(N))/(xNP1 - x(N)) - (u(N) - u(N-1))/(x(N) -...
        x(N-1)))/dx - (1/2)*(uNP1^2 - u(N-1)^2)/dx;
     rho_sm=Rho(y);
     for i = 2:N-1
         g(i+N) = (rho_sm(i+1) + rho_sm(i))*(x(i+1) - x(i)) - ...
               (rho_sm(i) + rho_sm(i-1))*(x(i) - x(i-1));
     end
     g(1+N) = (rho_sm(2) + rho_sm(1))*(x(2) - x(1)) - (rho_sm(1) + rho_sm(1))*
     (x(1) -x0);
```

```
     g(N+N) = (rho_sm(N) + rho_sm(N))*(xNP1 - x(N)) - (rho_sm(N) + rho_sm(N-1))*
     (x(N) - x(N-1));
     g(1+N:end) = - g(1+N:end)/(2*tau);
     rh = mass(y,N)\g;
end

%%%%%%%%%%%%%%%%%%% mass matrix %%%%%%%%%%%%%%%%%%%%%
function rh = mass(y)
    N1=length(y);
    N=N1/2;
    u = y(1:N);
    x = y(N+1:end);
    % fixed boundary values
    x0 = 0;
    u0 = 0;
    xNP1 = 1;
    uNP1 = 0;
    M1 = speye(N);
    M2 = sparse(N,N);
    M2(1,1) = - (u(2) - u0)/(x(2) - x0);
        for i = 2:N-1
            M2(i,i) = - (u(i+1) - u(i-1))/(x(i+1) - x(i-1));
        end
    M2(N,N) = - (uNP1 - u(N-1))/(xNP1 - x(N-1));
    M3 = sparse(N,N);
    e = ones(N,1);
    M4 = spdiags([e -2*e e],-1:1,N,N);
    rh = [M1 M2
          M3 M4];
end
%%%%%%%%%%%%%%%%%%%%%%% End mass %%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%% Mesh Density Function %%%%%%%%%%%%%%%%%%%%%
```

```
    %%%%%%%% Curvature mesh density function %%%%%%%%%

function rh=Rho(y)
   neq=length(y);
   N=neq/2;
   u = y(1:N);
   x = y(N+1:end);
   rho = zeros(N,1);
   v = zeros(N,1);
     for j=2:N-1
         v(j)=2/(x(j+1)-x(j-1))*( (u(j+1)-u(j))/(x(j+1)-x(j))-(u(j)-u(j-1))/(x(j)-
         x(j-1)) );
     end
   v(1) = 2*((x(2)-x(1))*(u(3)-u(1))-(x(3)-x(1))*(u(2)-u(1)))/((x(3)-x(1))*(x(2)-
   x(1))*(x(3)-x(2)));
   v(N) = 2*((x(N-1)-x(N))*(u(N-2)-u(N))-(x(N-2)-x(N))*(u(N-1)-u(N)))/((x(N-2)-
   x(N))*(x(N-1)-x(N))*(x(N-2)-x(N-1)));
   rho = rho + v.^2;
   % alpha calculation
   gamma = 1/3;
   Alpha = 0.0;
   for j=2:N
       Alpha = Alpha + (1/2)*(rho(j)^gamma+rho(j-1)^gamma)*(x(j)-x(j-1));
   end
   Alpha = (Alpha)^(3);
   % curvature mesh density function
   rho = (1+(1/Alpha)*rho).^(1/3);
   % smoothing mesh density function
   rho_sm=zeros(N,1);
   for j=2:(N-1)
      rho_sm(j) = 1/4*(rho(j-1)+rho(j+1))+1/2*rho(j);
   end
   rho_sm(1) = 1/2*(rho(1)+rho(2));
   rho_sm(N) = 1/2*(rho(N)+rho(N-1));
```

```
   rh=rho_sm;
end

     %%%%%%%% Arclegnth mesh density function %%%%%%%%%
function rh=Rho(y)
    N1 = length(y);
    N=N1/2;
    u = y(1:N);
    x = y(N+1:end);
    x0 = 0;
    u0 = 0;
    xNP1 = 1;
    uNP1 = 0;
    M = zeros(N,1);
    for i = 2:N-1
       M(i) = sqrt(1 + ((u(i+1) - u(i-1))/(x(i+1) - x(i-1)))^2);
    end
    M0 = sqrt(1 + ((u(1) - u0)/(x(1) - x0))^2);
    M(1) = sqrt(1 + ((u(2) - u0)/(x(2) - x0))^2);
    M(N) = sqrt(1 + ((uNP1 - u(N-1))/(xNP1 - x(N-1)))^2);
    MNP1 = sqrt(1 + ((uNP1 - u(N))/(xNP1 - x(N)))^2);

    % Spatial smoothing with gamma = 2, p = 2.

    SM = zeros(N,1);
    for i = 3:N-2
      SM(i) = sqrt((4*M(i-2)^2 + 6*M(i-1)^2 + 9*M(i)^2 + 6*M(i+1)^2 +
      4*M(i+2)^2)/29);
    end
    %SM0 = sqrt((9*M0^2 + 6*M(1)^2 + 4*M(2)^2)/19);
    SM(1) = sqrt((6*M0^2 + 9*M(1)^2 + 6*M(2)^2 + 4*M(3)^2)/25);
    SM(2) = sqrt((4*M0^2 + 6*M(1)^2 + 9*M(2)^2 + 6*M(3)^2 + 4*M(4)^2)/29);
    SM(N-1) = sqrt((4*M(N-3)^2 + 6*M(N-2)^2 + 9*M(N-1)^2 + 6*M(N)^2 +
    4*MNP1^2)/29);
```

```
    SM(N) = sqrt((4*M(N-2)^2 + 6*M(N-1)^2 + 9*M(N)^2 + 6*MNP1^2)/25);
    %SMNP1 = sqrt((4*M(N-1)^2 + 6*M(N)^2 + 9*MNP1^2)/19);
    rh = SM;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%% matlab ode15s solution %%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function rh=ode15s_sol(T,N)
    h = 1/(N+1);
    % initial condition & guess
    xint = h*(1:N)';
    uint = sin(2*pi*xint) + (1/2)*sin(pi*xint); % given initial condition
    u0=[uint; xint];
    tspan=[0 T];
    % option setting
    opts = odeset('RelTol',1e-14,'AbsTol',1e-12,'Mass',@mass_ode,'MaxOrder',5);
    sol= ode15s(@f_ode,tspan,u0,opts);
    y = deval(sol,T);
rh=y;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% main code %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Function Name : main                                                  %
% Description:The function called will generate the Tables 5.1-5.8 %
% by default the main function is set to generate the Table 5.1.   %
% In order to generate Table 5.2, we need to replace the function  %
% call ridc_fe with ridc_be.
% Inputs:                                                               %
% p = order of the method. In our case p = 4                            %
% t_int = start time                                                    %
% t_final = final time                                                  %
% y0 = initial condition                                                %
```

```
% Output : A table containing errors and orders                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function rh = main(p,t_int,t_final,y0)
  tspan=[t_int t_final];
  % step size
  a=0.01;
  K=p;
  % time step is halved
  dt_ar=[a;a/2;a/4;a/8];
  n=length(dt_ar);
  % error storing
  er=zeros(n,1);
  % dimension of vector
  N=length(y0);
  errors=zeros(n,1);
  orders_st=zeros(n,p);
  errors_st=zeros(n,p);
  % exact solution at the right most point
  u_ex=zeros(N,1);
  u_ex(:,1)=exp(t_final);  %  u_ex is defined from the exact solution given for
  % corresponding example, for the examples 5.3 and 5.6 exact solutions are
  % considered the solutions obtained from the subroutine ode15s_sol
  myf = @(t,x) f1(t,x);     % for example 5.1, for examples 5.2, 5.3, 5.4, 5.5
  % and 5.6 f1 is replaced with f2, f3, f4, f5 and f6 respectively.
  m=1;
  while (m<=p)
     for k=1:n
       dt=dt_ar(k);
       u_st =ridc_fe(myf,m,y0,tspan,dt,K);
       er(k)=norm( abs(u_st(:,end)-u_ex),inf);
     end
   p1=log2(er(n-3)/er(n-2));
   p2=log2(er(n-2)/er(n-1));
   p3=log2(er(n-1)/er(n));
```

```
    orders=[0;p1;p2;p3];
    errors=er;
    orders_st(:,m)=orders(1:n);
    errors_st(:,m)=errors(1:n);
    m=m+1;
 end
% craeting talex table
    Er=errors_st;
    Or=orders_st;
    format short g
    digits(4)

  A=zeros(n,2*p);

  A(:,1)=Er(:,1);
  A(:,2)=Or(:,1);

  A(:,3)=Er(:,2);
  A(:,4)=Or(:,2);

  A(:,5)=Er(:,3);
  A(:,6)=Or(:,3);

  A(:,7)=Er(:,4);
  A(:,8)=Or(:,4);

  digits(4)
  result=latex(sym(vpa(A)))

end
%%%%%%%%%%%%%%%%%%%%%%%% End main %%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%% End %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

%%

A.2 C++ Code

For the extension of the existing RIDC library to the moving mesh case we add the following member functions to the class **ImplicitOde** of the RIDC library. RIDC software and the user guidlines can be found from the link: http://mathgeek.us/software.html.

```
class ImplicitOde : public ODE
   {
  public:
  ImplicitOde(int my_neq, int my_nt, double my_ti, double my_tf, double my_dt)
     {
    neq = my_neq;
    nt = my_nt;
    ti = my_ti;
    tf = my_tf;
    dt = my_dt;
    }
/////////////////////////////////////////////////////////////////////////////////
/////////////////////////// Mesh density function ///////////////////////////////
/////////////////////////////////////////////////////////////////////////////////
void Rho(double *y, double *rho_sm)

  {
   int N=neq/2;

   vector<double> u(N);
   vector<double> x(N);

   for (int j=0;j<N;j++) {
       u[j] =y[j];
       x[j] =y[j+N]; }
```

```
vector<double> rho(N);

vector<double> v(N);

for (int j=1;j<N-1;j++) {
   v[j]=2.0/(x[j+1]-x[j-1])*( (u[j+1]-u[j])/(x[j+1]-x[j])-(u[j]-u[j-1])/(x[j]
   -x[j-1]) ); }

   v[0] = 2.0*((x[1]-x[0])*(u[2]-u[0])-(x[2]-x[0])*(u[1]-u[0]))/((x[2]-x[0])*
   (x[1]-x[0])*(x[2]-x[1]));
   v[N-1] = 2.0*((x[N-2]-x[N-1])*(u[N-3]-u[N-1])-(x[N-3]-x[N-1])*(u[N-2]-
   u[N-1]))/((x[N-3]-x[N-1])*(x[N-2]-x[N-1])*(x[N-3]-x[N-2]));

for (int j=0;j<N;j++) {
rho[j]=rho[j]+pow(v[j],2); }

// alpha calculation

double gamma = 1.0/3.0;

double Alpha =0.0;

for (int j=1;j<N;j++) {
   Alpha=Alpha+ 0.5*( pow(rho[j],gamma) + pow(rho[j-1],gamma) )*
   (x[j]-x[j-1]); }

 Alpha = pow(Alpha,3);

 vector<double> rh(N);

 for (int j=0;j<N;j++) {
    rh[j]=pow( (1.0+(1.0/Alpha)*rho[j]),(1.0/3.0));  }

// smoothing mesh density function
```

```
    for (int j=1;j<N-1;j++) {
       rho_sm[j] = 0.25*( rh[j-1]+rh[j+1] )+0.5*rh[j]; }

   rho_sm[0] = 0.5*(rh[0]+rh[1]);

   rho_sm[N-1] = 0.5*(rh[N-1]+rh[N-2]);

 }
////////////////////////////////////////////////////////////////////////////
////////////////////////////////// mass matrix /////////////////////////////
////////////////////////////////////////////////////////////////////////////

void mass(double t, double *y, double **M)

   {
     int N=neq/2;

     vector<double> u(N);
     vector<double> x(N);

     for (int j=0;j<N;j++) {
        u[j] =y[j];
        x[j] =y[j+N]; }

      /////// IC ///////

      double x0,u0,xNP1,uNP1;

      x0=0.0;
      u0=0.0;
      xNP1=1.0;
      uNP1=0.0;
```

```
vector< vector<double> > M1(N,vector<double>(N));
vector< vector<double> > M2(N,vector<double>(N));
vector< vector<double> > M3(N,vector<double>(N));
vector< vector<double> > M4(N,vector<double>(N));
// M1
for (int i=0;i<N;i++)  {
    M1[i][i]=1; }
// M2
M2[0][0]= -(u[1] - u0)/(x[1] - x0);

M2[N-1][N-1]=- (uNP1 - u[N-2])/(xNP1 - x[N-2]);

for (int i=0;i<N-1;i++) {
  M2[i][i]= - (u[i+1] - u[i-1])/(x[i+1] - x[i-1]); }

// M3
for (int i=0;i<N;i++) {
   for (int j=0;j<N;j++) {
     M3[i][j]=0; } }
 // M4
 M4[0][0]=-2;
 M4[0][1]=1;
 for (int i=1;i<N-1;i++) {
    M4[i][i]=-2;
    M4[i][i-1]=1;
    M4[i][i+1]=1; }
 M4[N-1][N-1]=-2;
 M4[N-1][N-2]=1;

  for (int i=0;i<N;i++) {
     for (int j=0;j<N;j++) {
         M[i][j]=M1[i][j];
         M[i][j+N]=M2[i][j];
         M[i+N][j]=M3[i][j];
```

```
              M[i+N][j+N]=M4[i][j]; } }
   }
////////////////////////////////////////////////////////////////////////
/////////////////////////// Gaussian elimination ///////////////////////
////////////////////////////////////////////////////////////////////////

void gauss(double *F, double **J, double *xn)

  {
    vector< vector<double> > A(neq,vector<double>(neq+1));
    for (int j=0; j<neq; j++) {
       for (int k=0; k<neq; k++)  {
          A[j][k]=J[j][k]; } }

    for (int j=0; j<neq; j++)  {
       A[j][neq]=F[j]; }

    ///////// solving AX=B  //////////

    for (int i=0; i<neq; i++) {
        // Search for maximum in this column
       double maxEl = abs(A[i][i]);
       int maxRow = i;
       for (int k=i+1; k<neq; k++) {
          if (abs(A[k][i]) > maxEl) {
             maxEl = abs(A[k][i]);
             maxRow = k;  } }

    // Swap maximum row with current row (column by column)

       for (int k=i; k<neq+1;k++) {
       double tmp = A[maxRow][k];
       A[maxRow][k] = A[i][k];
       A[i][k] = tmp; }
```

```
    // Make all rows below this one 0 in current column

        for (int k=i+1; k<neq; k++) {
           double c = -A[k][i]/A[i][i];
           for (int j=i; j<neq+1; j++) {
              if (i==j) {
                 A[k][j] = 0; }
              else {
                 A[k][j] += c * A[i][j]; }
            } }
     }  // end i loop

    // Solve equation Ax=b for an upper triangular matrix A

    vector<double> x(neq);

    for (int i=neq-1; i>=0; i--) {
      x[i] = A[i][neq]/A[i][i];
      for (int k=i-1;k>=0; k--)  {
        A[k][neq] -= A[k][i] * x[i]; } }

    for (int k=0;k<neq; k++) {
      xn[k] = x[k]; }

  }

////////////////////////////////////////////////////////////////////////////
////////////////////////// rhs of the ode y'=L^{-1}g(t,y) //////////////////////
////////////////////////////////////////////////////////////////////////////

void rhs(double t, double *y, double *f)

    {
```

```
 int N=neq/2;

 double ep, tau, x0,u0,xNP1,uNP1;

 ep=0.001;

 tau=0.1;

 vector<double> u(N);
 vector<double> x(N);

 for (int j=0;j<N;j++) {
   u[j] =y[j];
   x[j] =y[j+N]; }
 /////// IC ///////
 x0=0.0;
 u0=0.0;
 xNP1=1.0;
 uNP1=0.0;
////////////////

 double *g = new double[neq];

 double dx;

 for (int i=1;i<(N-1);i++) {
    dx = x[i+1] - x[i-1];
    g[i] = (2.0*ep)/dx*( (u[i+1] - u[i])/(x[i+1] - x[i] ) - (u[i] -
    u[i-1])/(x[i] - x[i-1]) )- 0.5*(pow(u[i+1],2)-pow(u[i-1],2))/dx; }

    dx = x[1] - x0;

    g[0] = (2.0*ep)/dx*((u[1] - u[0])/(x[1] - x[0]) - (u[0] - u0)/(x[0] -
```

```
    x0)) - 0.5*(pow(u[1],2) - pow(u0,2))/dx;

    dx = xNP1 - x[N-2];

    g[N-1]=(2.0*ep)/dx*((uNP1 - u[N-1])/(xNP1 - x[N-1]) - (u[N-1] - u[N-2])/
    (x[N-1] - x[N-2]))/dx - 0.5*(pow(uNP1,2) - pow(u[N-2],2))/dx;

 double *rho_sm = new double[N];

 Rho(y,rho_sm);

 double *v = new double[N];

 for (int i=1;i<(N-1);i++) {
    v[i] =( (rho_sm[i+1] + rho_sm[i])*(x[i+1] - x[i]) - (rho_sm[i] +
     rho_sm[i-1])*(x[i] - x[i-1]) ); }

    v[0] =  ( (rho_sm[1] + rho_sm[0])*(x[1] - x[0]) - (rho_sm[0] +
     rho_sm[0])*(x[0] - x0) );

    v[N-1] =( (rho_sm[N-1] + rho_sm[N-1])*(xNP1 - x[N-1]) - (rho_sm[N-1] +
     rho_sm[N-2])*(x[N-1] - x[N-2]) );

for (int i=0;i<N;i++) {
   g[i+N]=-1.0/(2.0*tau)*v[i];}

 double **L = new double*[neq];

 for (int j=0;j<neq;j++) {
        L[j] = new double[neq]; }

 mass(t,y,L);  // calling mass matrix L(y)

 double *w = new double[neq];
```

```
      gauss(g,L,w);  // linear solve

      for (int i=0;i<neq;i++) {
         f[i]=w[i]; }

      for (int i=0; i<neq; i++) {
       delete [] L[i]; }

      delete [] L;

      delete [] g;

      delete [] rho_sm;

      delete w;

      delete v;

   }

////////////////////////////////////////////////////////////////////////////
//////////////////////  function in the form F(x)=0  ////////////////////////
////////////////////////////////////////////////////////////////////////////

    void fun(double t, double *p, double *u, double *Fn)

      {
        double* frh = new double[neq];

        rhs(t,p,frh);

        for (int j=0;j<neq;j++) {
           Fn[j]=p[j]-u[j]-dt*frh[j]; }
```

```
        delete [] frh;
     }

////////////////////////////////////////////////////////////////////////////
////////////////////// Numerical jacobian of F(x)=0 ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void jac(double t, double *x, double *xold, double **J)
   {
     double dx=0.00000001;

     for (int i=0;i<neq;i++)
        {
        vector<double> xx(neq);

        for (int j=0;j<neq;j++)
          {
          xx[j]=x[j];
          }
    xx[i]=x[i]+dx;

    double *xn = new double[neq];

    for (int j=0;j<neq;j++) {
        xn[j]=xx[j]; }

    double *fx = new double[neq];

    fun(t,x,xold,fx);

    double *fxx = new double[neq];

    fun(t,xn,xold,fxx);
```

```
     for (int k=0;k<neq;k++) {
        J[k][i]=(fxx[k]-fx[k])/dx; }

    delete [] fx;

    delete [] xn;

    delete [] fxx;

    }
  }
////////////////////////////////////////////////////////////////////////////
/////////////////////////// l2 norm of vector //////////////////////////////
////////////////////////////////////////////////////////////////////////////

    void l2_norm(double *w, int n,double *norm)
        {
          double accum = 0.0;
          for (int i = 0; i < n; ++i) {
            accum += w[i] * w[i]; }
          norm[0]=sqrt(accum);
        }
////////////////////////////////////////////////////////////////////////////
/////////////////////////// Newton's Solver ////////////////////////////////
////////////////////////////////////////////////////////////////////////////

void step(double t, double *u, double *unew)

  {
  int max_iter=100;

      double tol=0.00000001;
```

```
int it_count=0;

vector<double> p(neq);

    vector<double> z(neq);

    for (int j=0;j<neq;j++) {
      p[j]=u[j];  }

    double *w = new double[neq];

    for (int j=0;j<neq;j++) {
      w[j]=u[j]; }

   // Newton's iteration starts here!

     for (int l=0;l<max_iter;l++)
          {
  it_count=it_count+1;

           double *v = new double[neq];

           for (int j=0;j<neq;j++) {
               v[j]=p[j]; }

           double *Fn = new double[neq];

           fun(t,v,w,Fn);   // calling function F(X)=0

           double **Jn = new double*[neq];

             for (int j=0;j<neq;j++) {
                Jn[j] = new double[neq]; }
```

```
jac(t,v,w,Jn);   // calling jacobian

double *x=new double[neq];

gauss(Fn,Jn,x);  // calling linear solver for x=Jn/Fn

for (int j=0;j<neq;j++) {
 z[j]=v[j]-x[j]; }

double *dv=new double[neq];

for (int j=0;j<neq;j++) {
  dv[j]=abs(z[j]-v[j]);}

double *norm=new double[1];

l2_norm(dv,neq,norm);

if (norm[0]<tol) {
break; }

p=z;

delete [] x;

delete [] Fn;

delete [] v;

delete [] dv;

delete [] norm;

for (int i=0; i<neq; i++) {
```

```
            delete [] Jn[i]; }
            delete [] Jn;
       } // end newton loop l


     for (int j=0;j<neq;j++) {
      unew[j]=z[j]; }

     delete [] w;

  } // end step
};
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////// main ////////////////////////////////
////////////////////////////////////////////////////////////////////////////
// By default main function will produce the running time of an arbitrary order//
// RIDC-BE method in moving mesh                                              //

int main(int argc, char *argv[])

 {
  int start_s=clock();
  int order, nt,N;
  double *sol;

  if (argc != 4) {
    printf("usage: <executable> <order> <nt>  >  output_file\n");
    fflush(stdout);
    exit(1); }
  else  {
    order = atoi(argv[1]); // order of method
    nt = atoi(argv[2]); // number of time steps
    N = atoi(argv[3]); }  // number of spatial mesh points
```

```
int neq = 2*N;
int ti = 0;
int tf = 1;

double dt = (double)(tf - ti)/double(nt); // compute dt

// initialize ODE variable

ImplicitOde *ode = new ImplicitOde(neq,nt,ti,tf,dt);

 double h = 1.0/(N+1);

 double *xint = new double[N];

 double *uintv = new double[N];

 for (int i=0;i<N;i++)
     {
     xint[i]=h*(i+1);
     uintv[i]=sin(2.0*M_PI*xint[i]) + 0.5*sin(M_PI*xint[i]);
     }

 sol = new double[neq];

 for (int i=0;i<N;i++)
     {
     sol[i]=uintv[i];
     sol[i+N]=xint[i];
     }

// call ridc

ridc_be(ode, order, sol);
```

```
  for (int i = 0; i < neq; i++) {
   printf("%17.16f\n", sol[i]); }

  delete [] sol;
  delete [] xint;
  delete [] uintv;

printf("Time taken: %.12fs\n", (double)(clock() - start_s)/CLOCKS_PER_SEC);

}
```