

Parallel High-Order Time Integrators

Bradley Gadd-Speakman



Year 4 Project
Mathematical Sciences
Durham University
April 2022

Declaration

This piece of work is a result of my own work except where it forms an assessment based on group project work. In the case of a group project, the work has been prepared in collaboration with other members of the group. Material from the work of others not involved in the project has been acknowledged and quotations and paraphrases suitably indicated.

Bradley Gadd-Speakman

Thank you to my project advisor Hossein Amini-Kafiabad for the amazing support throughout my final year project and also to my close friends and family.

Abstract

In this project, we investigated and implemented various time integrators with the aim of developing a high-order method for numerically solving first order initial value problems (IVPs) in a parallelisable manner. The method we focused on is revisionist integral deferred correction (RIDC)[4], which consists of prediction steps calculated via a typical time integration method and tiered corrections found using an update formula. Initially these values are calculated sequentially, but once enough of them have been found, the correction values across these tiers can then be calculated in parallel. With large enough computation time, RIDC can have an arbitrarily high order of accuracy as it can be shown to be proportional to the total number of prediction and correction tiers. The computation time of such a method would normally be large but due to the parallelisable nature of the algorithm we can dedicate one CPU core to each of the prediction and correction tiers, thus mitigating the added computational burden of the extra corrections performed in this high-order method.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Time Integration Fundamentals | 3 |
| 2.1 | Classical Time Integrators | 3 |
| 2.2 | Order of Accuracy | 5 |
| 2.3 | Stability | 7 |
| 2.4 | Parallelism | 9 |
| 3 | Integral Deferred Correction | 10 |
| 3.1 | Derivation and Overview of IDC | 11 |
| 3.2 | Convergence and Stability of IDC | 17 |
| 4 | Revisionist Integral Deferred Correction | 23 |
| 4.1 | Overview of RIDC | 23 |
| 4.2 | Python Code Walkthrough | 27 |
| 4.3 | Validation of RIDC | 35 |
| 5 | Partial Differential Equations | 37 |
| 5.1 | Spectral Methods | 38 |
| 5.1.1 | The Fourier Spectral Method | 38 |
| 5.1.2 | The Pseudo-Spectral Method | 41 |
| 6 | Conclusion | 48 |
| | Bibliography | 49 |
| A | Fortran Code for Parallel RIDC | 50 |

Chapter 1

Introduction

Time integration is a fundamental component of numerical methods for solving IVPs described by ODEs or PDEs. Accurate and efficient time integration is crucial for obtaining reliable results in simulations of physical systems and other applications. However, a big issue with most classical time integrators is that they are inherently sequential processes and therefore cannot be parallelised very easily or at all.

This report primarily investigates revisionist integral deferred correction (RIDC) as a high-order time integration method for numerically solving first-order IVPs. RIDC is an adaptation of the integral deferred correction (IDC)[5] method that retains its high accuracy while also allowing the process to be parallelised. Many figures are included in this report to more visually demonstrate concepts related to these techniques, which have all been produced by myself, predominantly using Python's matplotlib package [7].

IDC is a predictor-corrector method, which uses an initial time integration technique to 'predict' the approximate values of the solution to an IVP. It then sequentially 'corrects' these prediction values, increasing the order of accuracy of the numerical solution, using an update formula which we derive later in the report. This method is a popular alternative to more classical time integrators but again suffers by not being parallelisable.

RIDC is a modified version of IDC which, following a brief sequential setup phase, can simultaneously calculate correction values at nodes that are staggered in time. This allows it to utilise parallel computing resources effectively, making it ideal for large-scale simulations of IVPs where computational ef-

iciency is critical. Consequently, RIDC offers a significant improvement in computational efficiency compared to IDC due to its parallelisation capabilities.

This report will begin by providing an overview of various classical time integration methods and their properties. Next, we will derive and evaluate IDC. We then move on to describe the modifications made to IDC that have led to RIDC, and explain why RIDC can be parallelised under this new framework. Additionally, we will explore how these methods are used to solve PDEs, which are commonly used to model physical phenomena such as fluid dynamics, heat transfer, and electromagnetism in numerous scientific and engineering applications.

Overall, this report will provide a comprehensive understanding of time integrators, with a particular focus on RIDC and its ability to enable efficient parallelisation for large-scale simulations of IVPs using staggered corrections.

Chapter 2

Time Integration Fundamentals

In this chapter we will discuss the fundamentals of time integration which includes some classical time integrators such as Euler's method and Heun's method. We also cover the concepts of order of accuracy and stability which are properties of a time integrator that we can analyse to determine how well the method performs. More detailed and varied examples of time integrators and their properties can be found in J.C. Butcher's book [2].

Time integrators can be placed into two categories, explicit and implicit methods. Explicit methods calculate the approximate value of the solution at a given point in time using the previously approximated values in time. Implicit methods calculate the approximate value of the solution at a given point in time by solving an equation which involves both this term and previously approximated values in time. This report is solely concerned with explicit time integration methods.

2.1 Classical Time Integrators

To begin our review of classical time integrators, we start with the Euler method, the most basic explicit method. The canonical form of an IVP which we want to solve numerically is,

$$u'(t) = f(u, t), \quad t \in [0, T], \quad u(0) = u_0 \quad (2.1)$$

where $u \in \mathbb{R}^n$, $f : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^n$. We will always assume that there exists a unique solution to this IVP and this is precisely what we denote by $u(t)$.

All numerical time integrators begin by discretising the temporal domain into a sufficiently large number of nodes $\{t_n\}_{n \in \mathbb{N}}$. For the entirety of this report, we only consider the most straightforward discretisation of using N equidistant points in time. The constant step-size between these points is given symbolically by h and the time steps are exactly $t_n = nh$.

The formula for the Euler method is derived from equation (2.1) using the first order Taylor series approximation of u expanded about t_n ,

$$u(t) = u(t_n) + (t - t_n)f(u(t_n), t_n) + O((t - t_n)^2).$$

After evaluating this equation at time $t = t_{n+1}$, the next time step after t_n , we clearly see that we can get an approximation for the value of u at a new time step using the previous value of u in time. We denote this new approximate value by \hat{u}_{n+1} and the old by \hat{u}_n . The forward Euler method is therefore given by,

$$\hat{u}_{n+1} = \hat{u}_n + hf(\hat{u}_n, t_n), \quad \hat{u}_0 = u_0. \quad (2.2)$$

Other examples of explicit methods are Heun's method and the classic Runge-Kutta method (RK4). These methods require more function evaluations and more storage for intermediate values, this means they require more computation time and larger memory requirements. The upside of these methods however is that they have better properties than Euler's method, as will be shown in sections **2.2** and **2.3**.

Heun's method calculates approximate solution values to equation (2.1) using the following formula:

$$\begin{aligned} \tilde{u}_{n+1} &= \hat{u}_n + hf(\hat{u}_n, t_n), \\ \hat{u}_{n+1} &= \hat{u}_n + \frac{h}{2} \left[f(\hat{u}_n, t_n) + f(\tilde{u}_{n+1}, t_{n+1}) \right]. \end{aligned} \quad (2.3)$$

This method can be interpreted as a combination of Euler's method and the trapezoidal rule, resulting in a new time integration technique with now two unique evaluations of function f required.

The classic Runge-Kutta method is similar to the previous two methods, except now the function f is required to be evaluated four times for each new approximation value. Its formula is given by:

$$\begin{aligned}
k_1 &= f(\hat{u}_n, t_n), \\
k_2 &= f\left(\hat{u}_n + h\frac{k_1}{2}, t_n + \frac{h}{2}\right), \\
k_3 &= f\left(\hat{u}_n + h\frac{k_2}{2}, t_n + \frac{h}{2}\right), \\
k_4 &= f(\hat{u}_n + hk_3, t_{n+1}), \\
\hat{u}_{n+1} &= \hat{u}_n + \frac{h}{6} \left[k_1 + 2k_2 + 2k_3 + k_4 \right].
\end{aligned} \tag{2.4}$$

2.2 Order of Accuracy

The first property of time integrators we mention is that of order of accuracy or order of convergence. Before this concept can be understood however, we must know what is meant by the term global error. Global error e_n at time t_n of a numerical integration scheme is defined by $e_n = u_n - \hat{u}_n$ where u_n is the true solution value and \hat{u}_n is the numerical solution value at this time. We say a method is convergent to the true solution of an IVP if $\lim_{h \rightarrow 0} \max_n |e_n| = 0$. The method will have an order of accuracy p if the absolute global error $|e_n|$ is bound by a constant multiplied by the step-size h to the p th power

$$|e_n| \leq Ch^p. \tag{2.5}$$

Therefore, the order of accuracy is a number which quantifies the rate of convergence to the true solution for any IVP as the step-size is reduced to zero. For this reason it is a pivotal metric in determining how well a time integration method performs, as we ideally want as high an order of accuracy as possible.

You can analytically show that the previously mentioned time integrators: Euler's method, Heun's method and RK4, have orders of accuracy 1, 2 and 4 respectively. To show proof of these results however is beyond the scope of this project and we instead opt to demonstrate this experimentally. Formal proofs of the orders of accuracy for these methods can be found in [2]. To empirically display these results we use convergence plots. A convergence

plot is a graph of $\log(|e_n|)$ against $\log(h)$, the gradient of such a plot we expect to be approximately constant of value p . The intuition behind this is driven directly by inequality (2.5).

The IVP we want to solve to obtain the convergence plots is the test equation with its standard initial condition given by

$$u'(t) = \lambda u(t), \quad u_0 = 1 \quad (2.6)$$

where in general we let $\lambda \in \mathbb{C}$. For the case of making convergence plots however, we set $\lambda = 1$ and therefore know that the true solution to this IVP is $u(t) = e^t$.

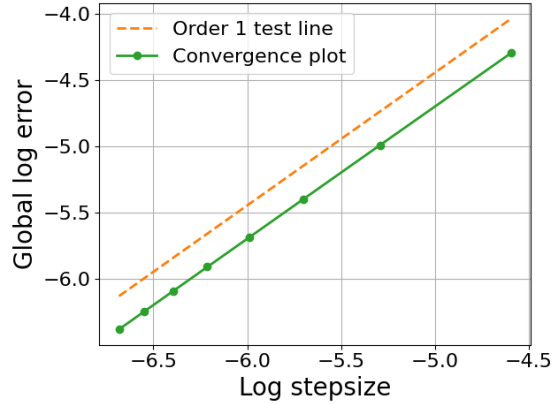


Figure 2.1: Convergence plot for the Euler method. The orange dashed reference line is perfectly straight with a gradient equal to the expected order of the method (in this case one).

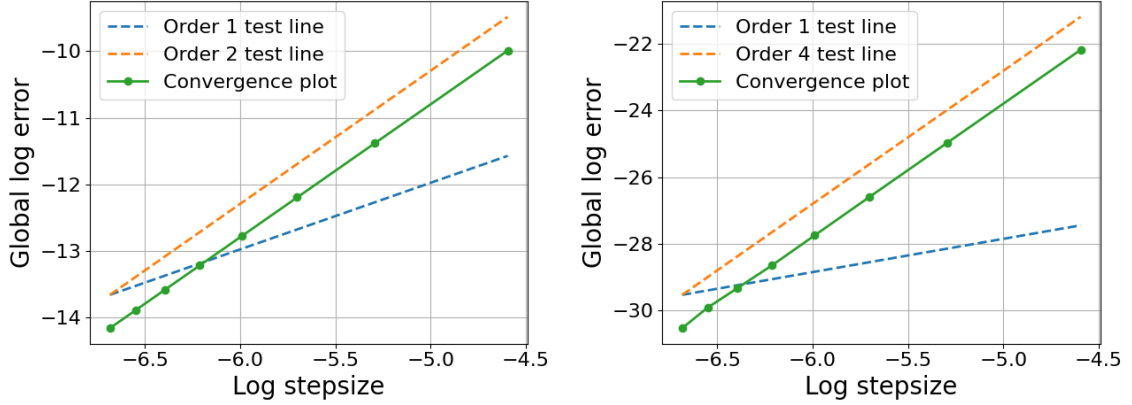


Figure 2.2: (left) The convergence plot for Heun's method. (right) The convergence plot for RK4.

2.3 Stability

The second property of time integrators we investigate is known as the stability region. We know the solution to the test equation (2.6) is exactly $u(t) = e^{\lambda t}$ and therefore if $\text{Re}(\lambda) < 0$ we have $\lim_{t \rightarrow \infty} u(t) = 0$. In this scenario we expect our numerical solution to (2.6) to get smaller in magnitude after a time step of size h . This motivates the following definition of the stability region S :

$$S = \{h\lambda : \text{Re}(\lambda) < 0 \text{ and } |\hat{u}_1| < 1\}. \quad (2.7)$$

The stability regions for all three previously mentioned explicit time integration methods can be found in the book [2] under the stability subsections for these techniques. The regions are given explicitly below:

- Euler's Method, $S = \{h\lambda : |1 + h\lambda| < 1\}$.
- Heun's Method, $S = \{h\lambda : \left|1 + h\lambda + \frac{(h\lambda)^2}{2}\right| < 1\}$.
- RK4, $S = \{h\lambda : \left|1 + h\lambda + \frac{(h\lambda)^2}{2} + \frac{(h\lambda)^3}{6} + \frac{(h\lambda)^4}{24}\right| < 1\}$.

It is useful to plot these regions so that comparisons of stability between the methods can be more easily made.

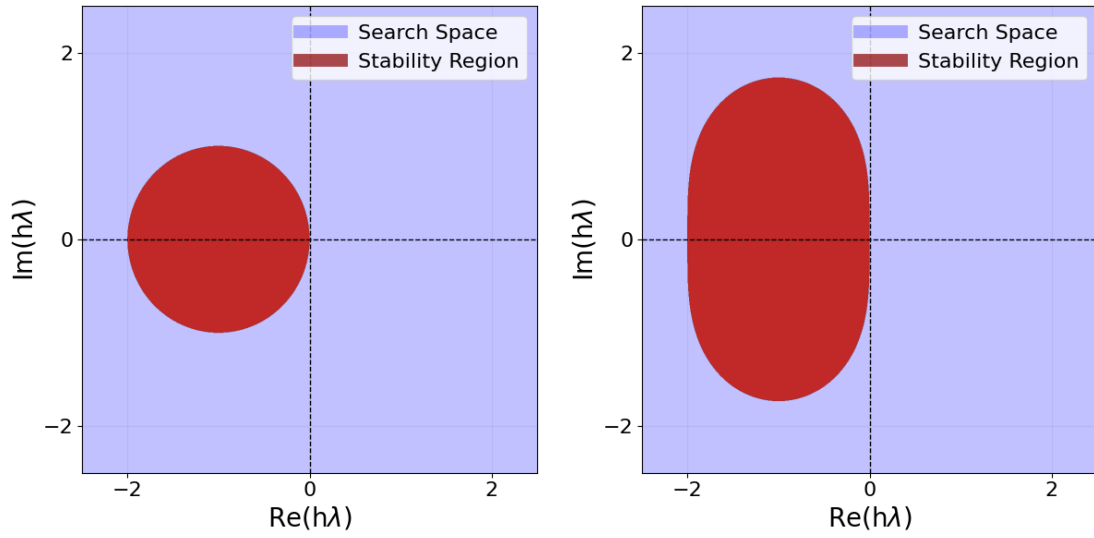


Figure 2.3: (left) The stability plot for Euler's method. (right) The stability plot for Heun's method.

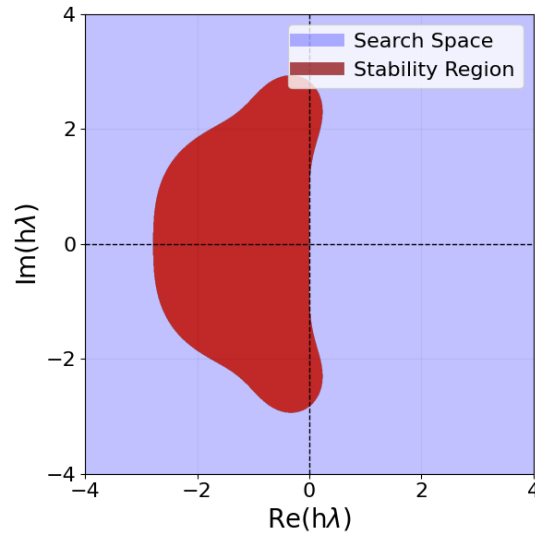


Figure 2.4: Stability plot for RK4.

Stability plots are useful for determining what pairs of values we can choose h and λ to be in order for the approximation to still converge to the true solution. We often want to use a step-size which is as large as possible, while still converging, in order to increase performance and that's why these plots are used. They demonstrate how lenient the methods are relative to one another in the acceptable pairs of values h and λ . From the stability plots, you can clearly see that RK4 has a much larger stability region than Euler or Heun's method.

2.4 Parallelism

All of the methods mentioned, and most time integrators in general, rely on a sequential calculation of new approximate solution values in time using previously calculated values. Time integration is, by its very nature, a sequential process and it is for this reason that it is conceptually difficult to imagine how a technique could possibly perform these calculations in parallel.

As will be shown in **chapter 4** on RIDC, we can get around this sequential limitation by using a method which incorporates multiple tiered corrections to our initial numerical solution values. As we increase the tier of correction, the order of accuracy provided also increases. We can then perform these calculations simultaneously (i.e. in parallel) over these so called correction levels.

Chapter 3

Integral Deferred Correction

The integral deferred correction (IDC) method discretises the time domain slightly differently to the previously seen techniques. IDC first divides the domain into J time intervals of equal size which are then further subdivided into K equidistant nodes in time. There are said to be a total of N nodes across the whole domain. This discretisation scheme is demonstrated below.

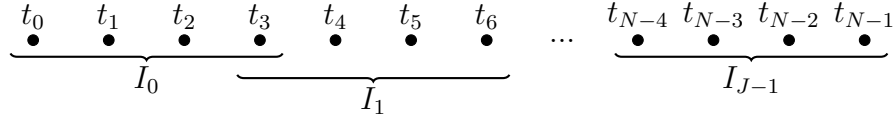


Figure 3.1: Enumeration of the time nodes for the IDC method with $K = 4$ nodes in each time interval.

IDC is a predictor-corrector method which performs iteratively over the time intervals. An initial time integration technique with the order of accuracy P is used to 'predict' the approximate values of the solution to an IVP. It then 'corrects' all these values for each of the M correction levels sequentially. All the predictions and corrections are calculated in the current time interval before moving on to the next. Once the final correction value in the current time interval is calculated, it is dropped down to all lower correction levels to be used as the initial condition for the next time interval.

These corrections are performed using an update formula also of order P derived from the error equation [4],

$$\frac{d\hat{u}^{(m+1)}}{dt} = \left(\int_0^t \hat{f}(\hat{u}^{(m)}, s) ds \right)' + f(\hat{u}^{(m+1)}, t) - f(\hat{u}^{(m)}, t). \quad (3.1)$$

Where $\hat{u}^{(m)}$ denotes the numerical solution after m corrections, f is the RHS to the canonical form of an IVP (2.1) and h is the step-size. The integral in (3.1) is approximated using a quadrature with integration points given by a subset of the nodes in the current time interval. **If at least $P(M+1)$ points are used to calculate the quadrature then IDC produces a numerical solution which has an order of accuracy of $P(M+1)$** [5]. Therefore, for simplicity, the rest of this chapter only considers IDC with $K = P(M+1)$ which results in $N = J(P(M+1) - 1) + 1$ and $h = \frac{T}{N-1}$.

3.1 Derivation and Overview of IDC

Before we can derive the error equation (3.1) we must first define two metrics, the error and the residual function.

The error function ϵ is given by $\epsilon(t) = u(t) - \hat{u}(t)$. This function quantifies the difference between the true and numerical solution.

The residual function δ is given by $\delta(t) = \hat{u}_0 + \int_0^t f(\hat{u}, s) ds - \hat{u}(t)$. This function quantifies how poorly our numerical solution satisfies the differential equation.

The following steps acquire the error equation:

We start by replacing $u(t)$ in $\epsilon(t)$ with its Picard integral form,

$$\begin{aligned} \epsilon(t) &= \hat{u}_0 + \int_0^t f(u, s) ds - \hat{u}(t) \\ &= \hat{u}_0 + \int_0^t f(\hat{u} + \epsilon, s) ds - \hat{u}(t). \end{aligned}$$

We introduce $\delta(t)$ into $\epsilon(t)$,

$$\epsilon(t) = \delta(t) + \int_0^t f(\hat{u} + \epsilon, s) - f(\hat{u}, s) ds.$$

Now we take the time derivative of both sides,

$$\epsilon'(t) = \delta'(t) + f(\hat{u} + \epsilon, t) - f(\hat{u}, t).$$

The term $\delta'(t)$ is equivalent to,

$$\delta'(t) = \left(\int_0^t f(\hat{u}, s) ds \right)' - \hat{u}'(t).$$

Resulting in,

$$\epsilon'(t) = \left(\int_0^t f(\hat{u}, s) ds \right)' - \hat{u}'(t) + f(\hat{u} + \epsilon, t) - f(\hat{u}, t).$$

We let $\hat{u}(t) = \hat{u}^{(m)}$, $\epsilon(t) = \epsilon^{(m)}$, $\hat{u}^{(m+1)} = \hat{u}^{(m)} + \epsilon^{(m)}$ and, after adding $\frac{d\hat{u}^{(m)}}{dt}$ to both sides, we get the error equation,

$$\frac{d\hat{u}^{(m+1)}}{dt} = \left(\int_0^t \hat{f}(\hat{u}^{(m)}, s) ds \right)' + f(\hat{u}^{(m+1)}, t) - f(\hat{u}^{(m)}, t).$$

The function f in the term $\left(\int_0^t f(\hat{u}^{(m)}, s) ds \right)'$ is approximated by an interpolating polynomial from the values of f evaluated using the $P(M+1)$ previous correction level values and time nodes inside the current time interval. To make this clear, the notation for the interpolation of f is given by \hat{f} . The error equation can be interpreted more easily after integrating both sides between t_n and t_{n+1} ,

$$\begin{aligned} g(\hat{u}^{(m+1)}, t) &= f(\hat{u}^{(m+1)}, t) - f(\hat{u}^{(m)}, t), \\ \hat{u}_{n+1}^{(m+1)} &= \hat{u}_n^{(m+1)} + \overbrace{\int_{t_n}^{t_{n+1}} g(\hat{u}^{(m+1)}, t) dt}^{\text{Order P Integration Method}} + \overbrace{\int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt}^{\text{Quadrature}}. \end{aligned} \quad (3.2)$$

We now derive update formulae for orders $P = 1, 2$:

Starting with $P = 1$, by Euler's method (2.2) we have,

$$\int_{t_n}^{t_{n+1}} g(\hat{u}^{(m+1)}, t) dt = h [f(\hat{u}_n^{(m+1)}, t_n) - f(\hat{u}_n^{(m)}, t_n)].$$

Then using (3.2) results in the $P = 1$ update formula

$$\hat{u}_{n+1}^{(m+1)} = \hat{u}_n^{(m+1)} + h [f(\hat{u}_n^{(m+1)}, t_n) - f(\hat{u}_n^{(m)}, t_n)] + \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt. \quad (3.3)$$

For $P = 2$, by Heun's method (2.3) we have,

$$\begin{aligned} \int_{t_n}^{t_{n+1}} g(\hat{u}^{(m+1)}, t) dt &= \frac{h}{2} [g(\hat{u}_n^{(m+1)}, t_n) + g(\tilde{u}_{n+1}^{(m+1)}, t_{n+1})], \\ \tilde{u}_{n+1}^{(m+1)} &= \hat{u}_n^{(m+1)} + h [f(\hat{u}_n^{(m+1)}, t_n) - f(\hat{u}_n^{(m)}, t_n)] + \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt. \end{aligned}$$

We first find expressions for $g(\hat{u}_n^{(m+1)}, t_n)$ and $g(\tilde{u}_{n+1}^{(m+1)}, t_{n+1})$,

$$g(\hat{u}_n^{(m+1)}, t_n) = f(\hat{u}_n^{(m+1)}, t_n) - f(\hat{u}_n^{(m)}, t_n),$$

$$g(\tilde{u}_{n+1}^{(m+1)}, t_{n+1}) = f(\hat{u}_n^{(m+1)}, t_n) + h g(\hat{u}_n^{(m+1)}, t_n) + \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt, t_{n+1}) - f(\hat{u}_{n+1}^{(m)}, t_{n+1}).$$

Then using (3.2) results in the $P = 2$ update formula

$$\hat{u}_{n+1}^{(m+1)} = \hat{u}_n^{(m+1)} + \frac{K_1}{2} + \frac{K_2}{2} + \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt, \quad (3.4)$$

where

$$\begin{aligned} K_1 &= h [f(\hat{u}_n^{(m+1)}, t_n) - f(\hat{u}_n^{(m)}, t_n)], \\ K_2 &= h [f(\hat{u}_n^{(m+1)}, t_n) + K_1 + \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt, t_{n+1}) - f(\hat{u}_{n+1}^{(m)}, t_{n+1})]. \end{aligned}$$

The quadrature used to approximate the integral in an update formula is given by

$$I_n = \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt \approx \sum_i S_{ni} \hat{f}_i, \quad (3.5)$$

where the i are the indices for nodes in the current time interval and

$$S_{ni} = \int_{t_n}^{t_{n+1}} \prod_{j \neq i} \frac{t - t_j}{t_i - t_j} dt,$$

$$\hat{f}_i = f(\hat{u}_i^{(m)}, t_i).$$

Since we are using equidistant nodes across the whole time domain, the matrix S can be calculated once at the very beginning of IDC/RIDC and never has to be recalculated at any point during the method.

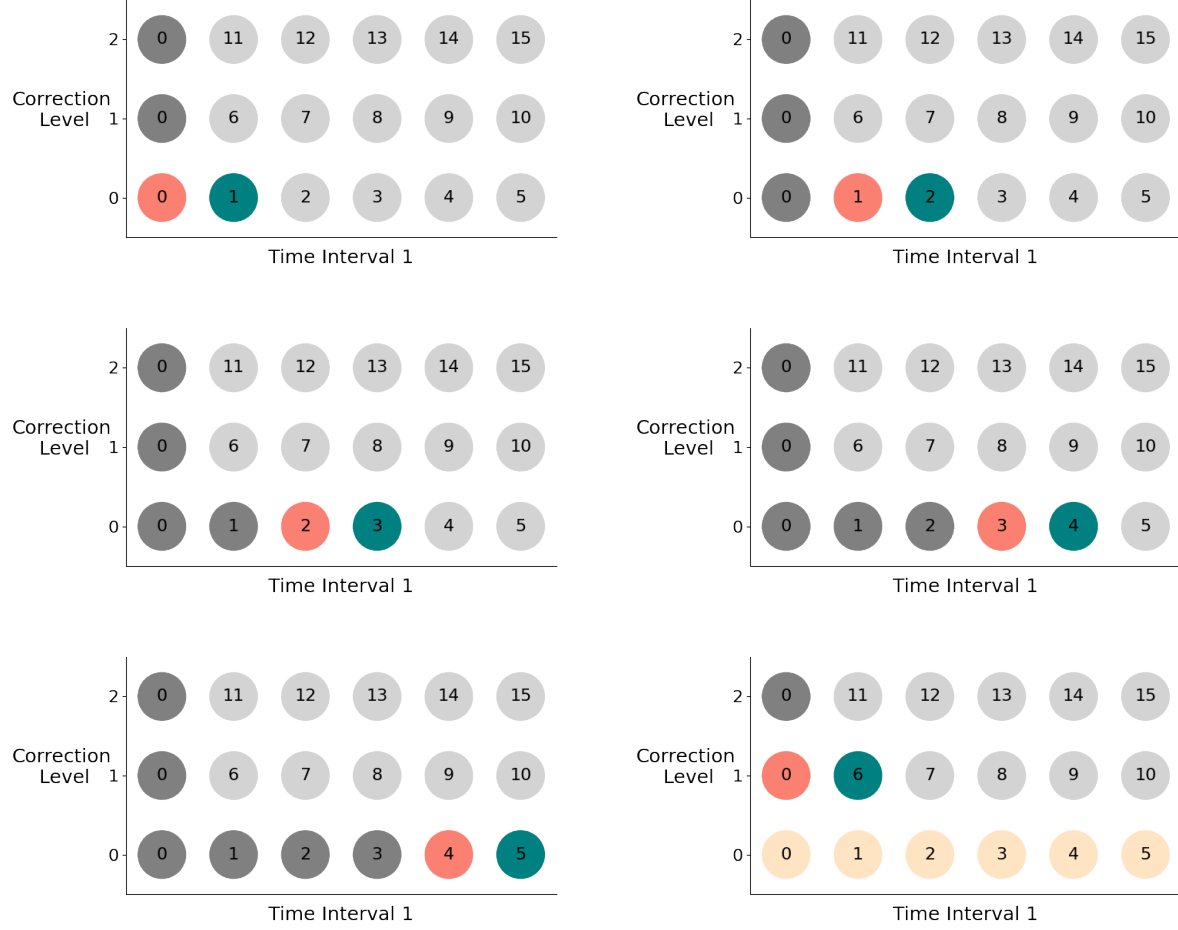
The order of calculation for values in the IDC method can be best visualised using a series of stencil diagrams. Nodes in the stencil diagrams are labelled with a number representing their order of calculation, they are also colour coded as follows:

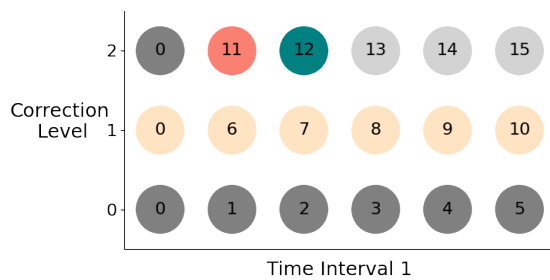
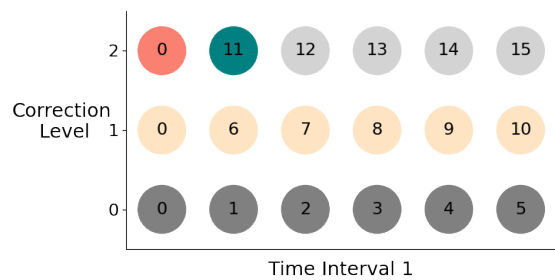
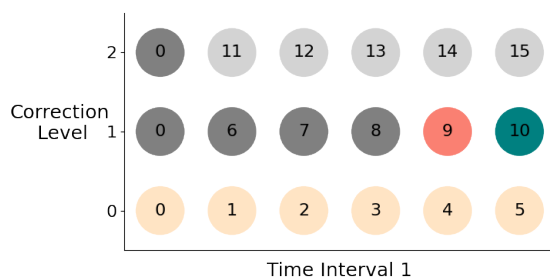
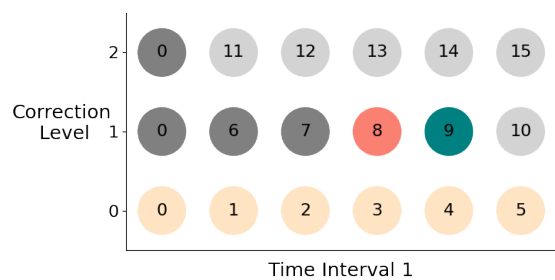
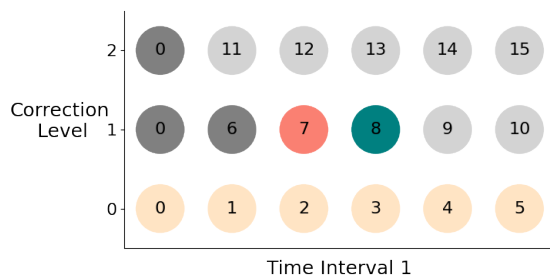
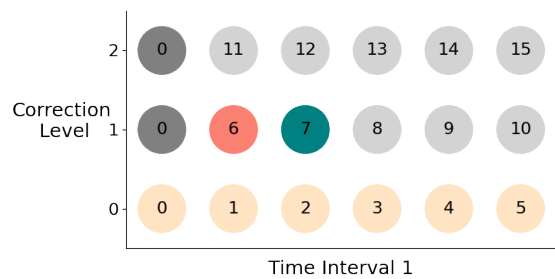
- Dark gray: Node which has already been calculated.
- Green: Current node being calculated.
- Red: Previous node in time, used to calculate the green node.
- Yellow: Nodes in the lower correction level, used to calculate I_n , the quadrature (3.5) in an update formula.

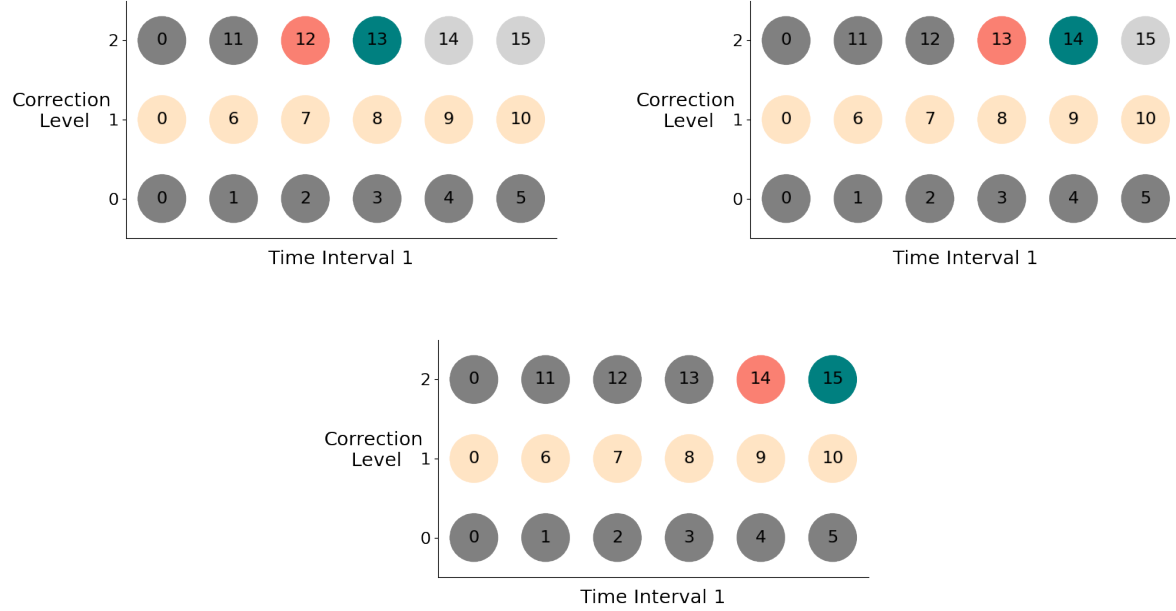
As an example of how this classification of nodes applies to an actual case, consider the $P = 1$ update formula (3.3). The values of a node with a certain color corresponds with the same coloured terms in the following formula,

$$\hat{u}_{n+1}^{(m+1)} = \hat{u}_n^{(m+1)} + h [f(\hat{u}_n^{(m+1)}, t_n) - f(\hat{u}_n^{(m)}, t_n)] + \int_{t_n}^{t_{n+1}} \hat{f}(\hat{u}^{(m)}, t) dt.$$

Stencil diagrams for the first time interval of the IDC method with parameters $P = 2$, $M = 2$ are shown below.







After the 15th node has been calculated, its value is dropped down to the lower corrections and acts as the initial condition for the second time interval. The order of calculated nodes is always the same for every interval.

It is important to note that IDC cannot be parallelised since all of the nodes are still calculated in a sequential manner. In order to make this a parallel algorithm, slight alterations must be made to the method. These changes lead us to RIDC which is covered in **chapter 4**.

3.2 Convergence and Stability of IDC

We now implement IDC, for various values of P and M , to inspect its order of accuracy which is proved in [5] to take the value $P(M + 1)$. The IVP we will be trying to solve is

$$u'(t) = 2ut, \quad t \in [0, T], \quad u_0 = 1, \quad (3.6)$$

which has the solution $u(t) = e^{t^2}$.

To obtain the convergence plots for IDC with Euler predictions and corrections i.e. $P = 1$, we used $T = 1$ and gradually decreased the constant step-size by iterating the number of time intervals $J = [16, 18, 20, 22, 24, 26]$.

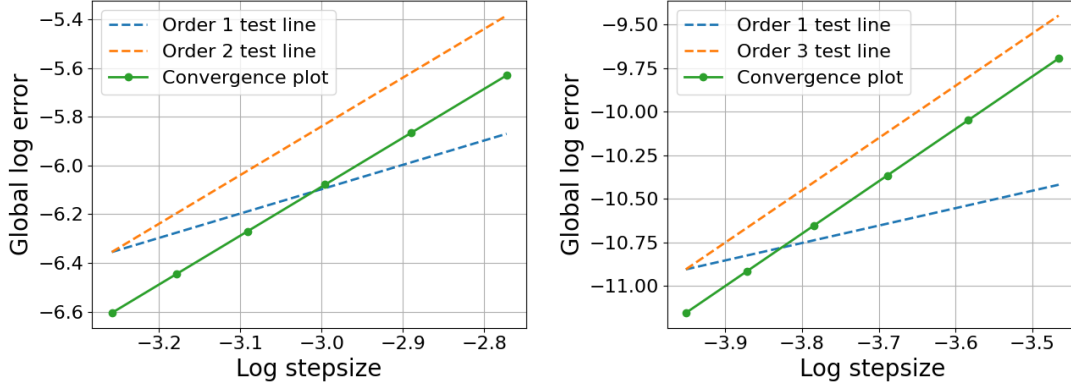


Figure 3.2: (left) IDC with $P = 1$, $M = 1$. (right) IDC with $P = 1$, $M = 2$.

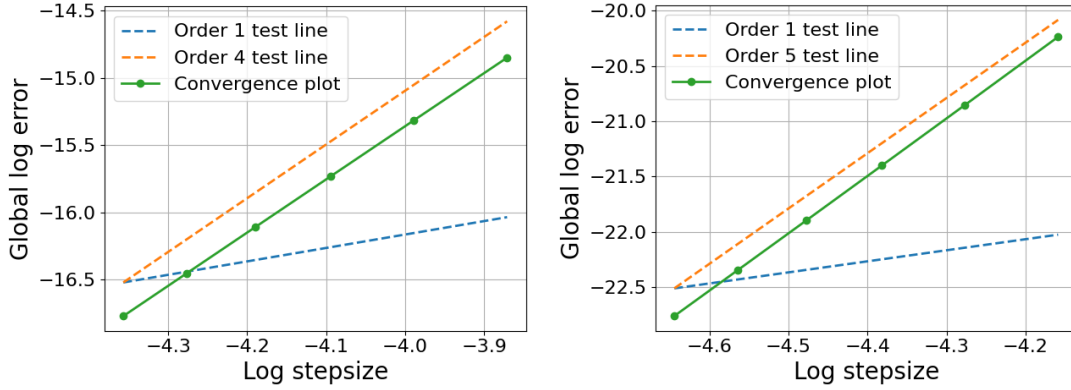


Figure 3.3: (left) IDC with $P = 1$, $M = 3$. (right) IDC with $P = 1$, $M = 4$. The narrowing of the space between the red and green lines at the top right of the right side graph suggests that the step-size is becoming too large for convergence.

To obtain the convergence plots for IDC with Heun predictions and corrections i.e. $P = 2$, we used a larger value $T = 4$ in order to forcefully increase the global error. This is done because otherwise the IDC method with $P = 2$ and $M = 4$ has an order of 10 which often makes global error so small that it reaches the resolution of machine epsilon, introducing huge rounding error in the calculations. The constant step-size was gradually decreased by iterating the number of time intervals $J = [16, 18, 20, 22, 24, 26]$.

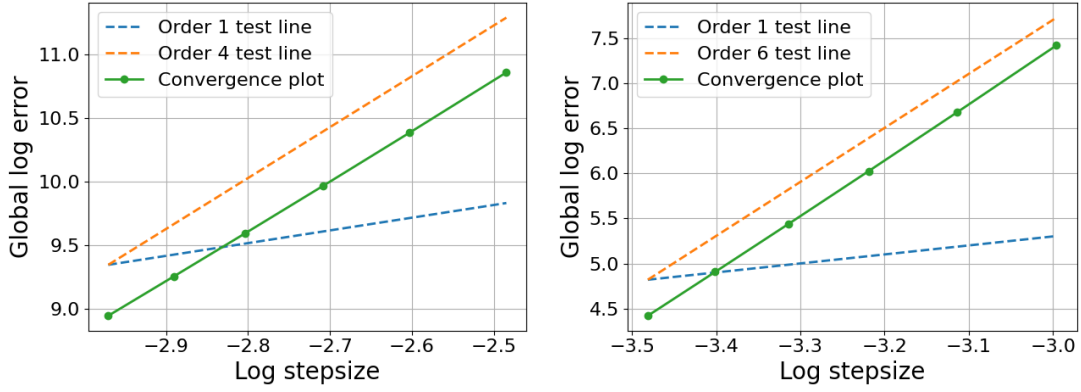


Figure 3.4: (left) IDC with $P = 2$, $M = 1$. (right) IDC with $P = 2$, $M = 2$.

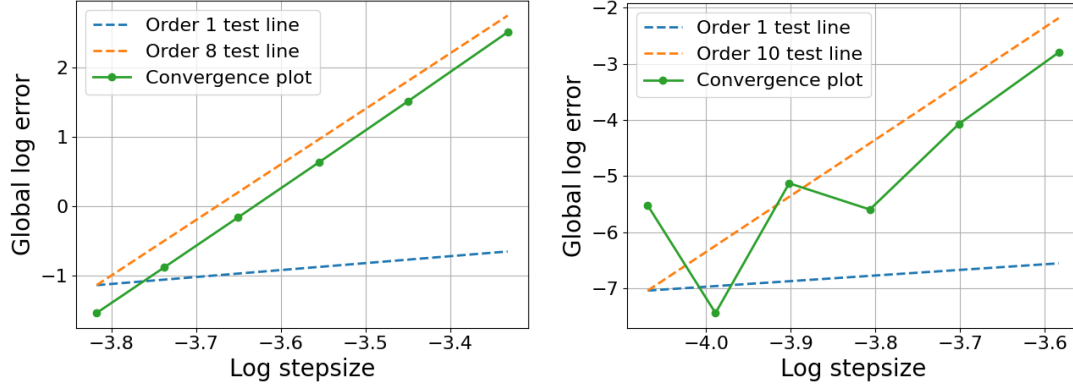


Figure 3.5: (left) IDC with $P = 2$, $M = 3$. (right) IDC with $P = 2$, $M = 4$. The instability at the start of the right side plot is most likely caused by the Runge phenomenon. Runge's phenomenon is the oscillatory inaccuracy of an interpolating polynomial which occurs at its boundaries when using high order polynomials and equidistant nodes.

Next we want to obtain stability regions for the IDC method. However, the stability region S given previously by (2.7) cannot be easily plotted for IDC/RIDC. This is because finding all the values $h\lambda$ which gives convergence of the test equation (2.6) is very difficult. So instead we use the following approximation of S :

$$S \approx \{\lambda : |\hat{u}_N| < 1\}, \quad (3.7)$$

where \hat{u}_N is the final value of the numerical solution to (2.6), $J = 1, T = P(M+1) - 1$. We use this value of T since this ensures the step-size is $h = 1$.

The following are stability plots for IDC with $P = 1, 2$:

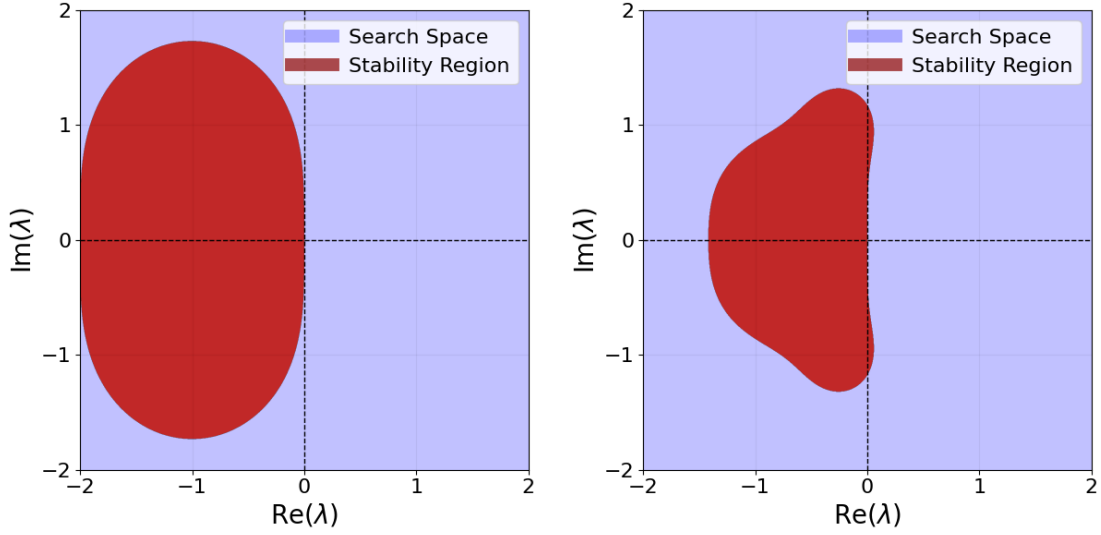


Figure 3.6: (left) $P = 1, M = 1$. (right) $P = 1, M = 2$.

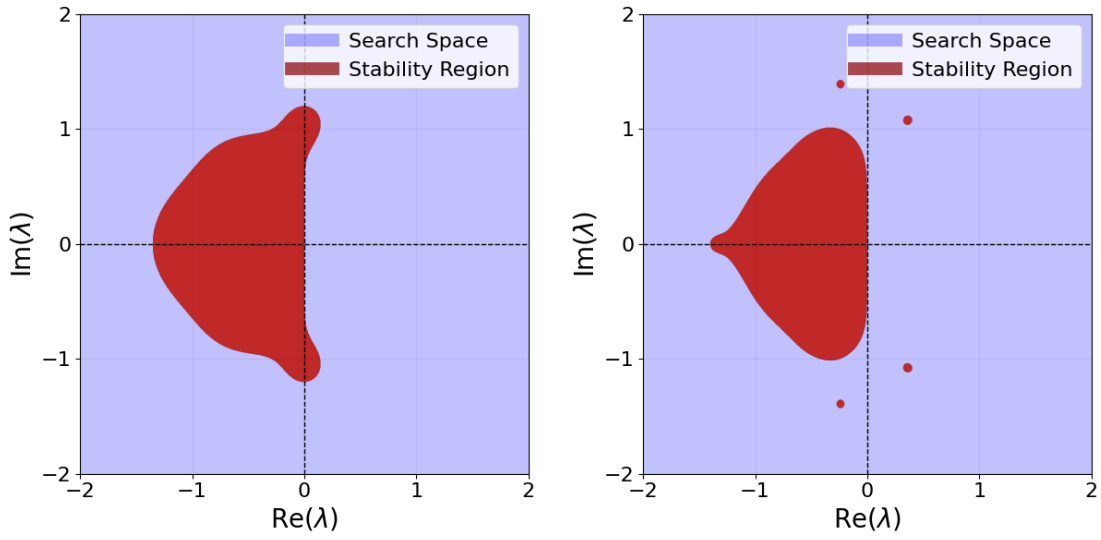


Figure 3.7: (left) $P = 1, M = 3$. (right) $P = 1, M = 4$.

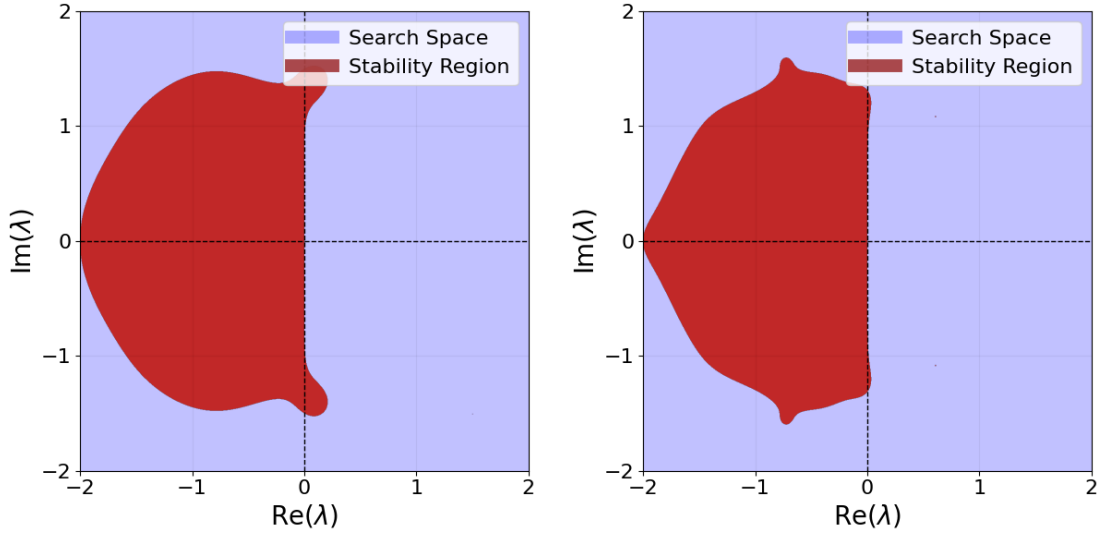


Figure 3.8: (left) $P = 2$, $M = 1$. (right) $P = 2$, $M = 2$.

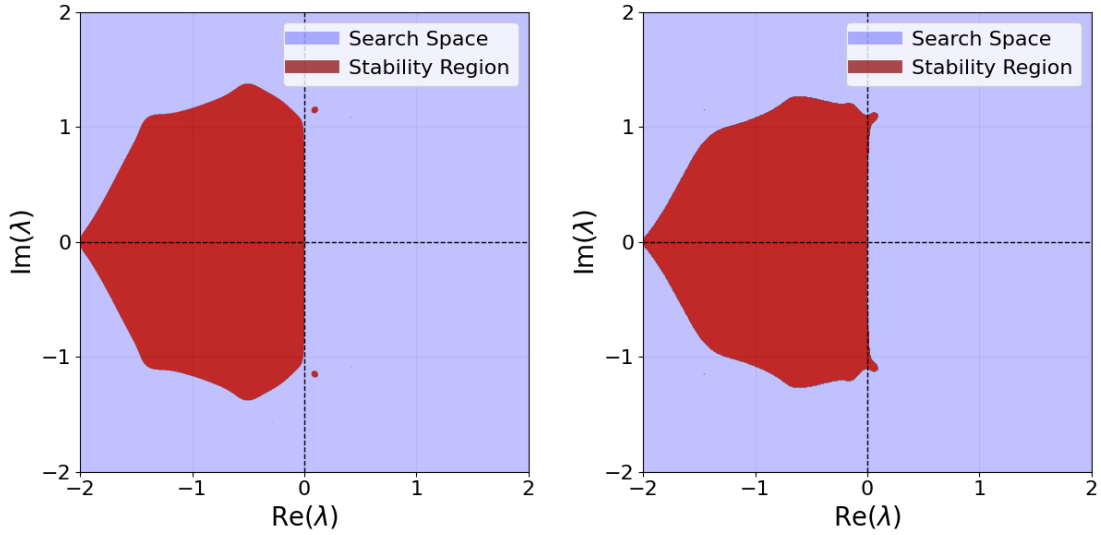


Figure 3.9: (left) $P = 2$, $M = 3$. (right) $P = 2$, $M = 4$.

From the stability plots we determine that higher order IDC methods will typically have better convergence for a wider range of parameters.

Chapter 4

Revisionist Integral Deferred Correction

Revisionist integral deferred correction (RIDC) is an adaptation of the IDC method which, for each time interval, allows the prediction and M correction levels to be calculated in parallel after an initial setup phase which runs sequentially. This theoretically allows for a speedup in computation time up to a factor of $M+1$, with a slight overhead inherent to all parallel algorithms. Each time interval still has K time nodes but the method is most optimal in terms of computation time when K is large. Since we use $K \gg P(M+1)$, the quadrature is no longer fixed to the first $P(M+1)$ nodes and instead advances in time. RIDC still incorporates the same update formulae as IDC and is proven in [4] to have an order of accuracy that is again given by $P(M+1)$.

4.1 Overview of RIDC

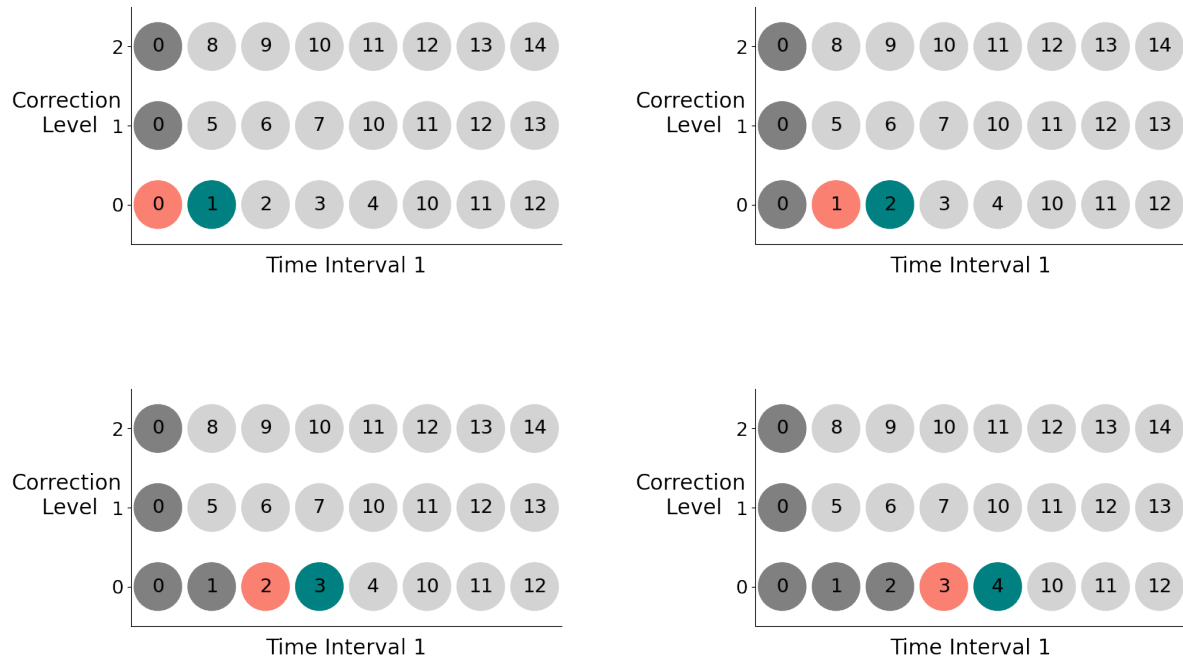
The setup phase has a specific order of calculations which creates a staggered structure of precalculated nodes. The nodes in this first phase are calculated in the following way. We iterate over the correction levels and, for each, only calculate the smallest number of nodes required such that the final level has $P(M+1)$ precomputed nodes.

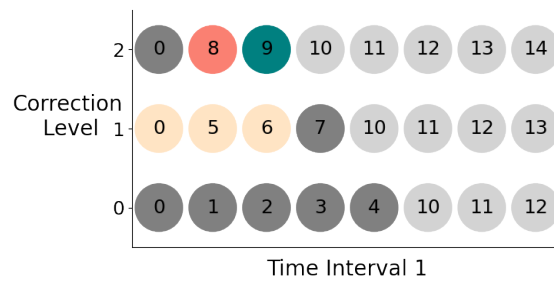
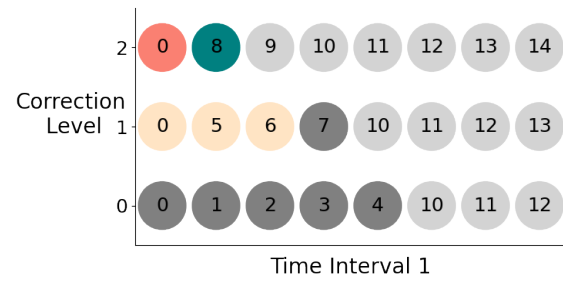
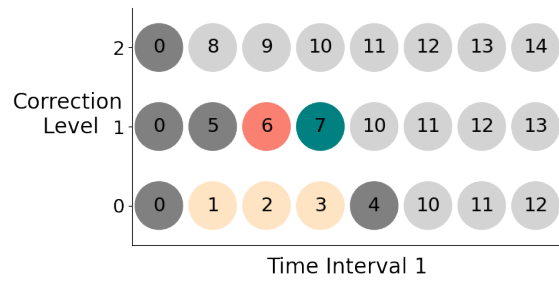
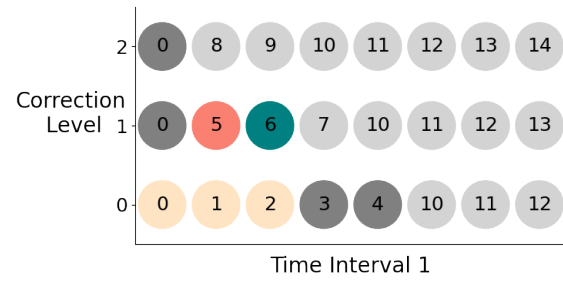
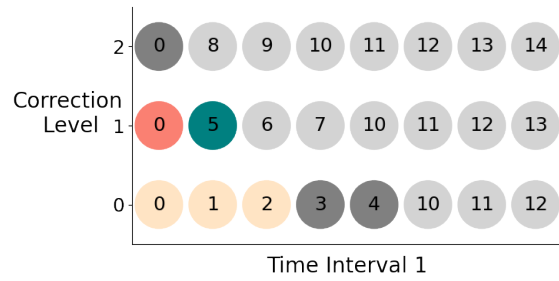
The parallel phase uses an update formula to simultaneously calculate prediction and correction values after the setup phase. This is possible because

the staggered node setup means that every correction level contains at least the $P(M + 1)$ nodes required for calculating the quadrature in the update formula. **A processor is assigned to each correction level to exclusively calculate values for that particular level. This is why the relative speedup of RIDC over IDC is approximately $M + 1$.**

Stencil diagrams for the RIDC method with parameters $P = 1, M = 2$ and $K = 8$ are displayed below:

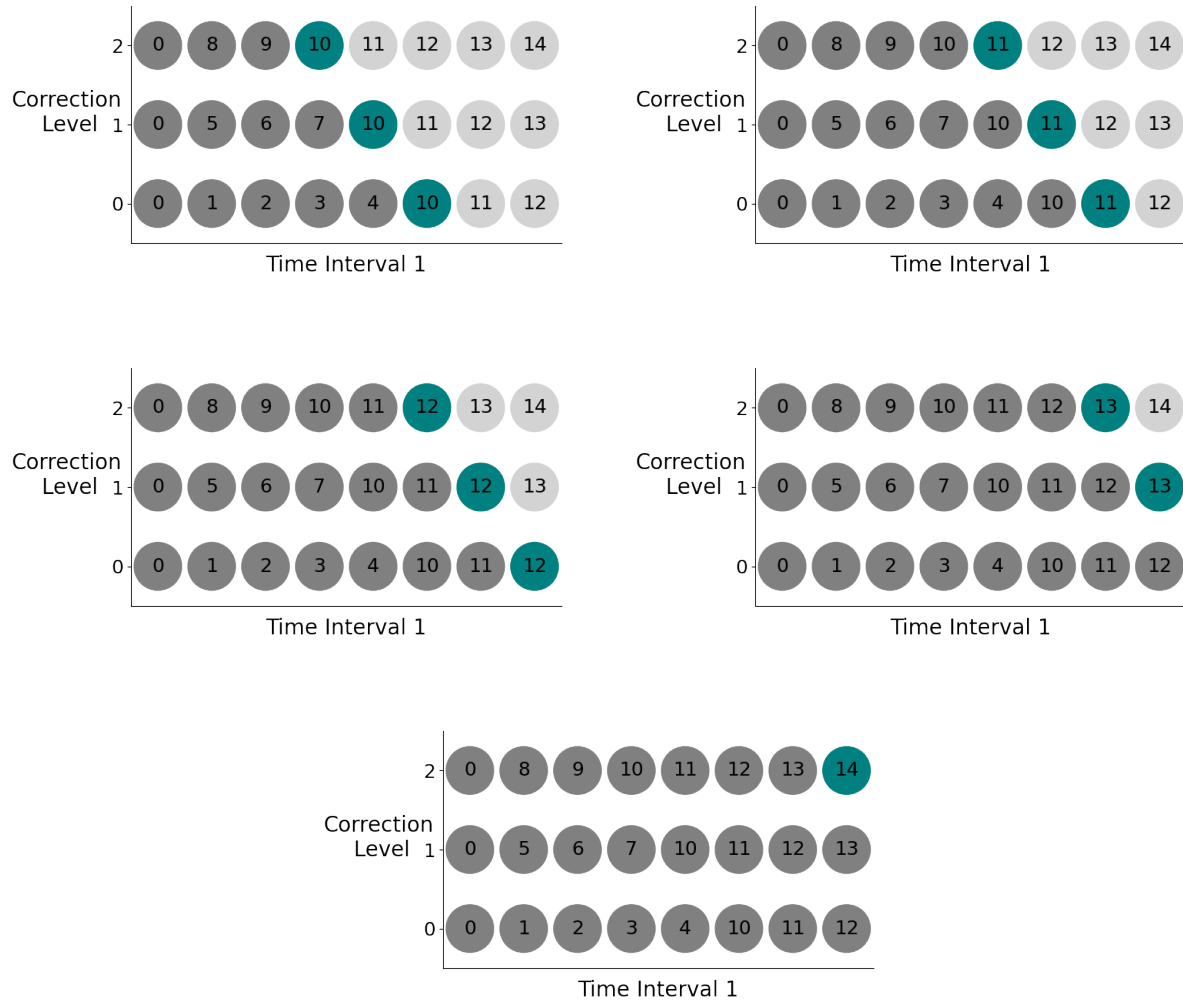
Setup Phase





Parallel Phase

The stencils in the parallel phase don't include the red or yellow node colouring since the previous correction values in time and the quadratures used can overlap. Since there is overlapping of these nodes, this means that RIDC must have some form of memory sharing between the processors running each of the correction levels.



Just like with IDC, once the 14th node has been calculated, its value is dropped down to the lower correction levels to be used as the initial condition for the next time interval.

4.2 Python Code Walkthrough

This subsection provides a detailed walkthrough of my own Python code for a sequential implementation of RIDC, enabling readers to gain a comprehensive understanding of the steps involved in the implementation of the RIDC method with $P = 1$.

Lines 1-31

```
1  import numpy as np
2
3  def RIDC_Euler(f:callable, u_0:float, T:float, J:int, K:int,
4                M:int, real:bool=True):
5      """
6          Performs RIDC vector time integration with Euler method
7          predictions and corrections.
8
9          Args:
10             f: Vector function f(u,t) on RHS of IVP.
11             u_0: Initial condition vector.
12             T: Upper bound on time domain.
13             J: Number of time intervals.
14             K: Number of nodes per interval. (K>>M)
15             M: Number of corrections.
16             real: Set to false if using complex numbers. (
17                 optional)
18
19             Returns:
20                 Final correction values for numerical solution across
21                 the whole time domain.
22             """
23
24             # Set default dtype for numpy
25             if real == True:
26                 data_type = np.float64
27             else:
28                 data_type = np.complex128
29
30             # Total number of nodes in time domain
31             N = J*(K-1)+1
32             # Get constant step size
33             h = T / (N-1)
34             # Get vector dimension
35             d = len(u_0)
```


- 1: Numpy is imported. This is the fundamental package for scientific computing in Python, it has features like multidimensional arrays, shape manipulation, linear algebra, etc [6]. It is also the only dependency used in the code.
- 3: The function 'RIDC_Euler' which will perform the RIDC method on an IVP is defined, along with its input arguments and their types.
- 4-18: The docstring for 'RIDC_Euler'.
- 20-31: If complex numbers are desired then the numerical solution will be stored using float64 real and float64 imaginary numbers. Important variables that will be used later are also initialised.

Lines 32-54

```

32
33     # Calculate quadrature weights matrix (uses M+1
integration points)
34     S = np.zeros(shape=(M, M+1), dtype=data_type)
35     t = np.linspace(0, M*h, num=M+1)
36     for i in range(M+1):
37         coef = np.zeros(shape=(M+1), dtype=data_type)
38         coef[-1] = 1
39         for j in list(range(0, i)) + list(range(i+1, M+1)):
40             coef = np.roll(coef, shift=-1)/(t[i]-t[j]) - coef
*t[j]/(t[i]-t[j])
41
42         for n in range(M):
43             S[n, i] = (
44                 sum(coef * t[n+1] ** np.arange(M+1, 0, step
=-1) / np.arange(M+1, 0, step=-1))
45                 -
46                 sum(coef * t[n] ** np.arange(M+1, 0, step=-1)
/ np.arange(M+1, 0, step=-1))
47             )
48
49     # Initialise initial condition
50     u_init = u_0
51
52     # Initialise numerical solution array
53     u_hat = np.zeros(shape=(N, d), dtype=data_type)
54     u_hat[0, :] = u_init

```

- 33-47: The quadrature weights matrix S used in (3.5) is calculated and stored. This only needs to be done once since we are using equidistant nodes, making the polynomial interpolation \hat{f} translationally invariant.
- 49-50: The variable `u_init` will hold the initial condition for the current time interval while the process iterates over them. Initially set to the initial condition of the IVP.
- 52-54: The numerical solution array `u_hat` is initialised.

Lines 55-80

```

55
56     # Loop over the time intervals
57     for j in range(J):
58         print(f"Time Interval: {j}")
59
60         #
61         -----
62         #
63         # Setup Phase: Sequentially calculate only the nodes
64         # necessary
65         # such that the final correction level has P(M+1)
66         # calculated
67         # nodes.
68
69         # Initialise array containing the first 2M-m+1
70         # vectors for each
71         # pred/corr level, shape: (corr level, time, vec dim)
72         u_first = np.zeros(shape=(M+1, 2*M+1, d), dtype=
73         data_type)
74
75         # Get initial condition
76         u_first[:, 0, :] = np.repeat([u_init], M+1, axis=0)
77
78         # First calculate predictions (Euler Method)
79         # Initialise start time
80         t = j*T/J
81         for i in range(2*M):
82             # Get prediction value
83             u_first[0, i+1, :] = u_first[0, i, :] + h*f(
84             u_first[0, i, :], t)
85             # Update time value
86             t += h

```

56-58: A for loop over the time intervals since the RIDC method acts sequentially on them.

66-71: Initialisation for the sequential setup phase which occurs at the start of every time interval. The variable `u_first` is created to contain the first $2M - m + 1$ values for each correction level m .

73-80: The prediction value is calculated using the Euler method.

Lines 81-117

```

81
82     # Now calculate corrections (Euler Update)
83     for m in range(1, M+1):
84         # Initialise start time
85         t = j*T/J
86         for i in range(2*M-m):
87             # Get quadrature offset and integration
88             # points indices
89             if i >= M:
90                 offset = M-1
91                 int_indices = range(i-(M-1), i+2)
92             else:
93                 offset = i
94                 int_indices = range(M+1)
95
96             # Get quadrature time nodes
97             quad_nodes = np.linspace(t-offset*h, t+(M-
98             offset)*h, num=M+1, dtype=np.float64)
99
100            # Get quadrature vector result
101            f_int = 0
102            for k in range(M+1):
103                f_int += (
104                    S[offset, k]
105                    *
106                    f(u_first[m-1, int_indices[k], :],
107                    quad_nodes[k])
108                )
109
110            # Get correction value
111            u_first[m, i+1, :] = (
112                u_first[m, i, :]
113                +
114                h*(f(u_first[m, i, :], t) - f(u_first[m

```

```

112         -1, i, :], t))
113         +
114         f_int
115     )
116     # Update time value
117     t += h

```

82-85: A for loop over the correction levels. The initial time value for the current correction level is initialised.

86-93: The indices for the nodes which are used in the quadrature are found and stored by `int_indices`. The variable `offset` denotes how many steps to the right of the first quadrature node the previous correction value node is.

95-105: Calculates the time values for each of the nodes in the quadrature. The quadrature value is then computed.

107-117: The correction value is calculated using the $P = 1$ update formula.

Lines 118-140

```

118
119     # Put the intervals first M+1, final correction
values
120     # into the solution array
121     u_hat[j*(K-1):j*(K-1)+M+1, :] = u_first[M, :M+1, :]
122
123     #
-----
124     #
125     # Parallel Phase: Simultaneously (ideally) calculate
the
126     # new correction values.
127
128     # Get array for holding the newly calculated values
in parallel
129     u_new = np.zeros(shape=(M+1, d))
130
131     # Get array containing the last M+1 values for each
of the
132     # pred/corr levels, last value is most recent.
133     # shape: (corr level, time, vec dim)
u_prev = np.zeros(shape=(M+1, M+1, d))

```

```

134         for m in range(M+1):
135             for i in range(M+1):
136                 u_prev[m, i, :] = u_first[m, i+(M-m), :]
137
138             # Can now delete the u_first array which is only used
139             # for setup
140             # to save memory manually
141             del u_first

```

119-121: The first $M + 1$ values for the final correction level are stored in the numerical solution matrix.

127-129: A matrix `u_new` is initialised which can contain all the new values for each correction level which are calculated simultaneously during the parallel phase. Since multiple levels need access to this simultaneously, this variable must be public and shared by all of them.

130-136: A new array `u_prev` is created which stores the $M + 1$ previously calculated values for each correction level. This variable must also be public.

138-140: The `u_first` variable is deleted to free up memory.

Lines 141-183

```

141
142         # This is the start of the section which can be
143         # parallelised
144         # Loop over the time step, we stop if i which
145         # represents the index
146         # of the current node for the final correction level
147         # is >= K.
148         for i in range(M, K-1):
149             # Prediction loop
150             if i+M < K-1:
151                 # Get current time
152                 t = j*T/J+(i+M)*h
153                 # Get prediction value
154                 u_corr = u_prev[0, -1, :] + h*f(u_prev[0, -1,
155                 :], t)
156
157                 # Add prediction value to new values array
158                 u_new[0, :] = u_corr
159
160             # Correction loops
161             for m in range(1, M+1):

```

```

157         if i+M-m < K-1:
158             # Get time for current loop at current
step in algorithm
159             t = j*T/J+(i+M-m)*h
160
161             # Get quadrature time nodes
162             quad_nodes = np.linspace(t-(M-1)*h, t+h,
num=M+1)
163
164             # Get quadrature value
165             f_int = 0
166             for k in range(M+1):
167                 f_int += (
168                     S[M-1, k]
169                     *
170                     f(u_prev[m-1, k, :], quad_nodes[k
171 ])
172                 )
173             # Get correction value
174             u_corr = (
175                 u_prev[m, -1, :]
176                 +
177                 h*(f(u_prev[m, -1, :], t) - f(u_prev[
m-1, -2, :], t))
178                 +
179                 f_int
180             )
181
182             # Add correction value to new values
array
183             u_new[m, :] = u_corr

```

142-144: This is where we would normally begin incorporating multi-threaded parallelism. There are now enough values calculated in each correction level such that the process can run simultaneously over them.

145-183: Much of this is similar to the setup phase, however there are some minor differences. An offset value is no longer necessary and all new values calculated are stored in the `u_new` array. Once the `u_new` array is filled for the current time step then the threads are allowed to continue to the next step simultaneously. This prevents a race condition when the `u_prev` array is updated.

Lines 184-201

```
184
185         # Put the final correction value into the
solution array
186         u_hat[j*(K-1)+i+1, :] = u_corr
187
188         # Here we must wait for all threads to arrive at
this
189         # point before continuing to update the previous
values
190         # array
191         u_prev[:, :-1, :] = u_prev[:, 1:, :]
192         u_prev[:, -1, :] = u_new
193
194         #
-----
195         #
196         # End Phase: Update the initial condition for the
next time
197         # interval
198
199         # Drop down final correction value to lower
correction levels
200         u_init = u_prev[M, -1, :]
201
202     return u_hat
```

185-186: This puts the final value calculated in the last correction level into the numerical solution array. This would not be feasible under parallelisation since `u_corr` wouldn't be guaranteed to represent the final correction level.

188-192: If this was a parallelised implementation then we would first have to wait here for all threads to catch up with one another before updating the `u_prev` array. This is important to do in order to avoid a race condition.

195-199: This is the end phase for the current time interval. The final value in the last correction level is used as the new initial condition for the next time interval.

201: The numerical solution array is returned as the output of the function.

4.3 Validation of RIDC

Although RIDC should have almost identical analytical properties to IDC, it is still important to ensure that the RIDC method has been programmed correctly. We use convergence plots to check that we obtain the expected order of accuracy, validating our implementation. To obtain the convergence plots, equation (3.6) is solved with $T = 4$. The RIDC method was initialised with parameters $P = 1$, $K = 100$ and the step-size h was gradually decreased by iterating over $J = [1, 2, 3, 4, 5]$.

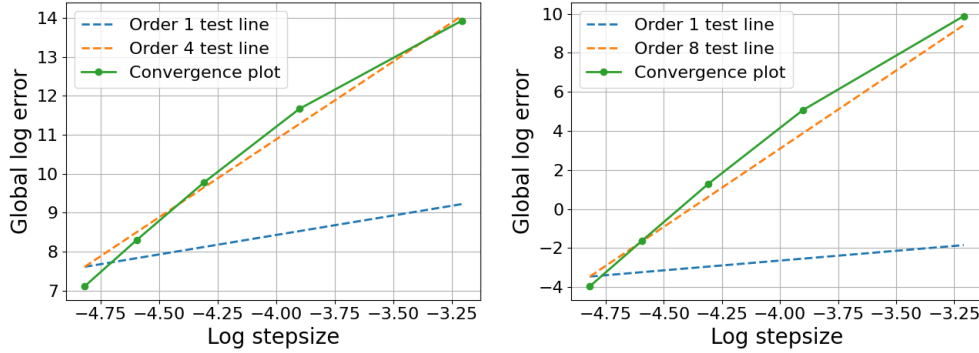


Figure 4.1: (left) RIDC with $P = 1$, $M = 3$, $K = 100$. (right) RIDC with $P = 1$, $M = 7$, $K = 100$. The plots agree with their expected orders of accuracy, 4 and 8 respectively.

We also implemented RIDC on a large 2D n-body problem using a parallelised and sequential version of the code in order to inspect the difference in computation time. A large n-body problem was chosen since the benefit of using parallel RIDC increases as f , the RHS to (2.1), becomes more computationally intensive to evaluate, making the benefit from parallelism as prominent as possible. For this comparison, my own source codes were programmed in Fortran and compiled using the Intel[©] Fortran Compiler. The parallel implementation uses the OpenMP library [3] to incorporate multi-threaded parallelism over the correction levels.

The 2D n-body problem with $n = 128$ bodies was,

$$x''_i(t) = 0.01 \sum_{j=1, j \neq i}^{128} \left(\frac{x_j - x_i}{\|x_j - x_i\|^3} \mathbb{1}(\|x_j - x_i\| > 0.01) \right) - x'_i(t),$$

where $x_i \in \mathbb{R}^2$, $i = 1, 2, \dots, 128$, $t \in [0, 1]$ and $\mathbb{1}$ is the indicator function.

This dynamical system is easily put into canonical form, turning this set of 128 second order ODEs into 256 first order IVPs,

$$\begin{cases} u'_i(t) = 0.01 \sum_{j=1, j \neq i}^{128} \left(\frac{x_j - x_i}{\|x_j - x_i\|^3} \mathbb{1}(\|x_j - x_i\| > 0.01) \right) - u_i(t), \\ x'_i(t) = u_i(t). \end{cases} \quad (4.1)$$

These 256 vector IVPs of dimension 2 were then flattened into a single vector IVP of dimension 512, which was then solved using the parallel and sequential versions of RIDC, 5 times each. Both of the integrators were setup with the same parameters; $P = 1$, $J = 10$, $K = 1000$, $M = 4$. The results for the total wall-clock time (in seconds) taken for RIDC to complete were as follows:

| Version | 1 | 2 | 3 | 4 | 5 | Mean |
|------------|----------|----------|----------|----------|----------|----------|
| Sequential | 350.5472 | 350.5016 | 351.3865 | 350.9930 | 351.1342 | 350.9125 |
| Parallel | 110.2889 | 102.8561 | 100.5951 | 97.9449 | 113.5925 | 105.0555 |

Based on the mean results from (4.1), I determine that my parallel implementation is a significant improvement on the sequential RIDC method, achieving an average speedup of approximately 3.3 times. While this value is still much below the theoretical maximum speedup of 5 times for this specific setup, an improved parallel implementation would be able to achieve a speedup closer to optimal. There are many plausible reasons for why my result is sub-optimal, a list of a few are given below:

- The function f may not be computationally intensive enough to benefit most from parallelism still.
- The value of K could be too low to benefit most from parallelism.
- My application of multi-threaded parallelism is surely sub-optimal.
- All parallel algorithms have inherent initialisation costs of threads/processors.
- CPU time may be a better comparison metric.
- My laptop could have been thermal throttling.

Chapter 5

Partial Differential Equations

Partial Differential Equations (PDEs) are an essential tool for mathematical modeling with broad applications in various fields, including physics, engineering, finance, and biology. PDEs involve multiple variables and their partial derivatives, modeling complex systems that vary continuously in space and time with several interacting variables. As such, finding analytical solutions to PDEs is a difficult task, and numerical methods provide a powerful tool for obtaining approximate solutions.

One common approach to finding numerical solutions of PDEs is to use time integration methods to obtain a sequence of solutions at discrete time points, and then combine these with spatial methods to obtain an approximate solution of the PDE at all points in the domain. The choice of time integration method and spatial discretisation method can have a significant impact on the accuracy and efficiency of the numerical solution, and a range of approaches have been developed, including finite difference, finite element, and spectral methods.

The main focus of this chapter is to explore the combination of RIDC with spectral methods as a technique for obtaining numerical solutions to PDEs. By combining these methods, we aim to achieve solutions with both high temporal and spatial order of accuracy within an optimised amount of computation time via parallelism.

5.1 Spectral Methods

Spectral methods are another class of techniques used to numerically solve differential equations, however they are typically used for those which are defined on a spatial domain. They all assume that the solution can be written as a linear combination of a set of predetermined basis functions. We then choose the coefficients which allow the sum to best satisfy the differential equation. The process for which these coefficients are found is what defines a specific spectral method. Spectral methods all have extremely high order of accuracy if the basis functions chosen are suitable to the problem. This is because they interpolate a function globally, rather than locally as in finite element schemes for example.

The general form of a spectral solution \hat{u} to a PDE involving variables in time and space is,

$$\hat{u}(x, t) = \sum_n c_n(t) \psi_n(x), \quad (5.1)$$

where the c_n are time dependent coefficients to be found and the ψ_n are predetermined basis functions.

5.1.1 The Fourier Spectral Method

As an initial example, we will use the spectral method to numerically solve the 1D heat equation, assuming plane wave basis functions. In other words, we are attempting to find the best Fourier series which satisfies the PDE,

$$\begin{aligned} \frac{\partial u}{\partial t} &= k^2 \frac{\partial^2 u}{\partial x^2}, \\ u(x + L, t) &= u(x, t), \quad u(x, 0) = u_0(x), \quad x \in [0, L], \quad t \in [0, T]. \end{aligned} \quad (5.2)$$

We begin by substituting (5.1) into (5.2) to obtain,

$$\sum_n c'_n(t) \psi_n(x) = k^2 \sum_n c_n(t) \psi''_n(x).$$

The plane wave basis functions are given by,

$$\psi_n(x) = \frac{1}{\sqrt{L}} e^{-i\alpha_n x}, \quad \alpha_n = (-1)^n \left\lceil \frac{n}{2} \right\rceil \frac{2\pi}{L}$$

Resulting in,

$$\sum_n (c'_n(t) + \alpha_n^2 c_n(t)) \psi_n(x) = 0.$$

We define the inner product by,

$$\langle f, g \rangle = \int_0^L f(x) \overline{g(x)} dx.$$

The plane wave basis functions are orthonormal in this Hilbert space and can therefore be used to find IVPs for the time dependent coefficients c_n ,

$$\langle \psi_n, \psi_m \rangle = \begin{cases} 1 & \text{if } n = m \\ 0 & \text{if } n \neq m \end{cases},$$

$$\left\langle \sum_n (c'_n(t) + \alpha_n^2 c_n(t)) \psi_n, \psi_m \right\rangle = \sum_n (c'_n(t) + \alpha_n^2 c_n(t)) \langle \psi_n, \psi_m \rangle = 0.$$

The IVPs for finding the c_n are then,

$$c'_n(t) + \alpha_n^2 c_n(t) = 0, \quad c_n(0) = \int_0^L u_0(x) \overline{\psi_n(x)} dx.$$

To make this process an algorithm, only N many basis functions are used in the numerical solution. We then use RIDC to solve for the N coefficient functions. The inner product integral first needs to be calculated to obtain the initial condition values and this is found using a quadrature. In this case, the quadrature we use is the discrete Fourier transform,

$$c_n(0) = \frac{1}{N} \sum_{j=0}^{N-1} u_0(x_j) e^{\frac{2\pi i n j}{N}},$$

where the x_j are equidistant nodes in $[0, L]$. We then retrieve N values of the solution (5.1), for each time step, evaluated at the nodes x_j by using the inverse Fourier transform on the coefficients found via RIDC. It is shown in [9] that the Fourier spectral method has an order of accuracy of N if the basis functions used are suitable for the problem being solved.

For our example (5.2), we use $T = 0.2$, $L = 2\pi$, $k = 1$ and the initial condition $u_0(x) = 2 \sin 3x + \sin 9x - 3 \cos 5x$. The joint RIDC, Fourier spectral solution is then acquired by performing the steps above.

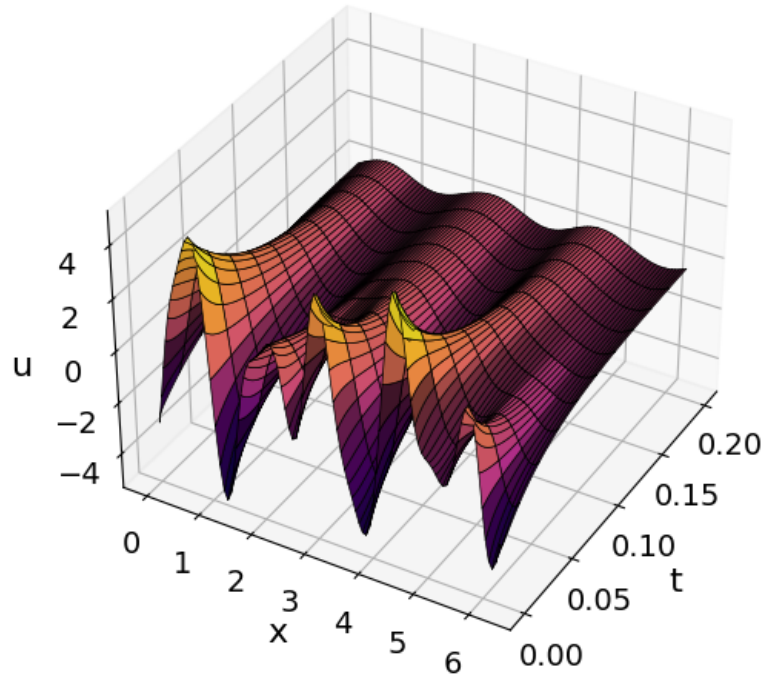


Figure 5.1: The numerical solution to (5.2) using RIDC time integration and the Fourier spectral method. The parameters used for the integration methods are: $N = 100$, $P = 1$, $J = 10$, $K = 1000$, $M = 3$. We therefore expect this solution to have an order of accuracy in time of 4 and in space of 100.

5.1.2 The Pseudo-Spectral Method

The Fourier spectral method encounters problems when trying to numerically solve more complicated PDEs involving non-linear terms. For example, solution values produced by these non-linear terms can exhibit non-continuous like behaviour, this is a problem for all global interpolation methods. It is also computationally costly to find the coefficients for the product of multiple functions where one is expressed as a known spectral expansion. This problem can be mitigated via an alteration to the Fourier spectral method known as the pseudo-spectral method.

We want to multiply two scalar functions $u(x)$ and $f(x)$ with $x \in \mathbb{R}^d$ where f is expressed as a known spectral expansion with the same N plane wave basis functions ψ_i in each of the d spatial dimensions. The resulting function is denoted h and we want to find the coefficients in its expansion.

$$h(x) = u(x)f(x), \quad x \in \mathbb{R}^d.$$

$$f(x) = \sum_{e_0, \dots, e_{d-1}=0}^{N-1} C_{e_0, \dots, e_{d-1}} \prod_{i=0}^{d-1} \psi_{e_i}(x_i), \quad C \text{ is known.}$$

$$h(x) = \sum_{e_0, \dots, e_{d-1}=0}^{N-1} \Phi_{e_0, \dots, e_{d-1}} \prod_{i=0}^{d-1} \psi_{e_i}(x_i), \quad \Phi \text{ is unknown.}$$

We can find the coefficients d-tensor Φ in the expansion of $h(x)$ by performing the inner product d times,

$$\Phi_{e_0, \dots, e_{d-1}} = \langle \dots \langle \langle h, \psi_{e_0} \rangle_0, \psi_{e_1} \rangle_1 \dots, \psi_{e_{d-1}} \rangle_{d-1}, \quad (5.3)$$

where the inner products are taken over each of the spatial dimensions individually,

$$\langle f, g \rangle_i = \int_a^b f(x_i) \overline{g(x_i)} dx_i,$$

resulting in,

$$\Phi_{e_0, \dots, e_{d-1}} = \sum_{m_0, \dots, m_{d-1}=0}^{N-1} \overbrace{\langle \dots \langle \langle u(x) \prod_{i=0}^{d-1} \psi_{m_i}(x_i), \psi_{e_0} \rangle_0, \psi_{e_1} \rangle_1 \dots, \psi_{e_{d-1}} \rangle_{d-1}} C_{m_0, \dots, m_{d-1}}.$$

Therefore we can find the coefficients tensor Φ by performing the tensor contraction given by,

$$\Phi_{e_0, \dots, e_{d-1}} = \sum_{m_0, \dots, m_{d-1}=0}^{N-1} V_{e_0, \dots, e_{d-1}, m_0, \dots, m_{d-1}} C_{m_0, \dots, m_{d-1}}. \quad (5.4)$$

Obvious issues with this method are that the 2d-tensor V must be precomputed and we also have to perform a computationally expensive tensor contraction. The properties of the plane wave basis functions and by extension V reveal the computational complexity of this operation.

We notice two important properties of the plane wave basis functions,

$$\overline{\psi_a(x_i)} = \psi_{-a}(x_i), \quad \psi_a(x_i) \psi_{-b}(x_i) = \psi_{a-b}(x_i).$$

The formula for the elements of V is given by,

$$\begin{aligned} V_{e_0, \dots, e_{d-1}, m_0, \dots, m_{d-1}} \\ = \int_a^b \cdots \int_a^b u(x) \psi_{m_0}(x_0) \overline{\psi_{e_0}(x_0)} dx_0 \cdots \psi_{m_{d-1}}(x_{d-1}) \overline{\psi_{e_{d-1}}(x_{d-1})} dx_{d-1}. \end{aligned}$$

We inspect the innermost integral of the previous formula and use the noted properties of the basis functions,

$$\begin{aligned} V_{e_0, m_0} &= \int_a^b u(x) \psi_{m_0}(x_0) \overline{\psi_{e_0}(x_0)} dx_0 \\ &= \int_a^b u(x) \psi_{m_0 - e_0}(x_0) dx_0 = v_{m_0 - e_0}. \end{aligned}$$

This means that the d $N \times N$ matrices given by the values in the e_i, m_i axes of the tensor V are Toeplitz matrices. The tensor contraction (5.4) used to find Φ is therefore a series of d nested discrete convolutions, each with computational complexity $O(N^2)$ [8]. The resulting computational complexity of finding the coefficients tensor Φ is therefore $O(N^{2d})$.

The pseudo-spectral method requires that the spatial domain is discretised into a d -dimensional grid of N^d equidistant points. We use d -tensors H, U, F to denote the functions h, u, f evaluated at each of the points in the grid, this form of notation is known as discrete variable representation (DVR).

$$H = U \odot F,$$

where $(\cdot \odot \cdot)$ is the element-wise multiplication operator. The pseudo-spectral method uses a d -dimensional discrete Fourier transform (DFT) as a quadrature which directly approximates the nested inner products in equation (5.3) in order to find the coefficients tensor Φ of H . This removes the requirement to precompute the tensor V as in equation (5.4).

$$\Phi_{e_0, \dots, e_{d-1}} = \sum_{m_0=0}^{N-1} \cdots \sum_{m_{d-1}=0}^{N-1} H_{m_0, \dots, m_{d-1}} \prod_{i=0}^{d-1} \psi_{e_i}(-\frac{2\pi m_i}{N}).$$

Directly using this discrete transform allows us to take advantage of the d -dimensional fast Fourier transform (FFT) which can be interpreted as performing d nested 1-dimensional FFTs, each of which requires a number of computations scaling with $N \log N$ [1]. Therefore, the pseudo-spectral method also reduces the computational complexity of finding the coefficients tensor Φ from $O(N^{2d})$ to $O((N \log N)^d)$.

To illustrate the pseudo-spectral method, we try to solve the 2-dimensional incompressible vorticity equation with no external forces,

$$\begin{aligned} \frac{\partial w}{\partial t} &= \nu \nabla^2 w - (\mathbf{u} \cdot \nabla) w, \quad -\nabla^2 \mathbf{u} = \nabla \times \mathbf{w}, \\ w(x+L, y+L, t) &= w(x, y, t), \quad w(x, y, 0) = w_0(x, y), \\ x, y &\in [0, L]^2, \quad t \in [0, T]. \end{aligned} \tag{5.5}$$

We first apply the 2-dimensional FFT to (5.5) and obtain,

$$\begin{aligned} \frac{\partial \tilde{w}}{\partial t} &= - \mathcal{F} \left\{ \overbrace{u_x \mathcal{F}^{-1} \{ i k_x \tilde{w} \}}^{\text{Function Mult.}} + \overbrace{u_y \mathcal{F}^{-1} \{ i k_y \tilde{w} \}}^{\text{Function Mult.}} \right\} \\ &\quad - \nu (k_x^2 + k_y^2) \tilde{w}. \end{aligned}$$

To find the coefficients of the bracketed terms, which highlight the function multiplication that the pseudo-spectral method is concerned with, we could perform a convolution but instead use the FFT. This reduces the computational complexity of finding the coefficients of $\frac{\partial w}{\partial t}$ from $O(N^4)$ to $O((N \log N)^2)$.

We find that u_x and u_y are sole functions of the coefficients \tilde{w} using $-\nabla^2 \mathbf{u} = \nabla \times \mathbf{w}$ and noticing,

$$\tilde{u}_x = \frac{k_y}{k_x^2 + k_y^2} i \tilde{w}, \quad \tilde{u}_y = -\frac{k_x}{k_x^2 + k_y^2} i \tilde{w}.$$

Resulting in,

$$\begin{aligned} \frac{\partial \tilde{w}}{\partial t} = & -\mathcal{F}\{\mathcal{F}^{-1}\{\tilde{u}_x\}\mathcal{F}^{-1}\{ik_x \tilde{w}\} + \mathcal{F}^{-1}\{\tilde{u}_y\}\mathcal{F}^{-1}\{ik_y \tilde{w}\}\} \\ & - \nu(k_x^2 + k_y^2)\tilde{w}. \end{aligned} \quad (5.6)$$

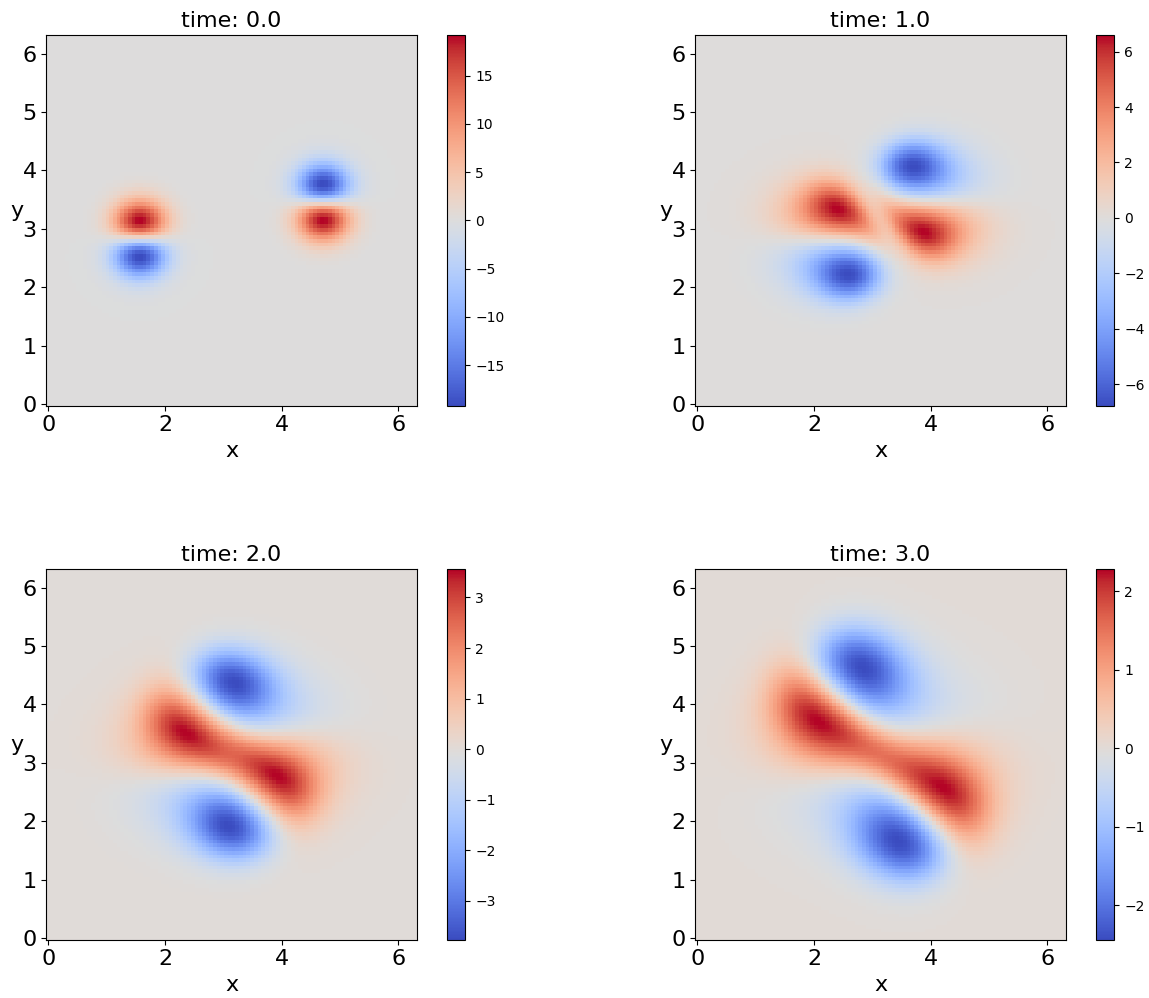
Therefore, to find our numerical solution we need to solve for \tilde{w} by using RIDC on (5.6) with the initial condition $\tilde{w}_0 = \mathcal{F}\{w_0\}$. We then use the inverse FFT algorithm to obtain our solution values of interest, w from \tilde{w} .

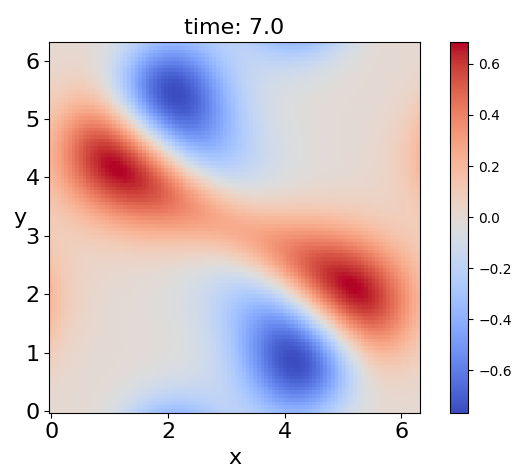
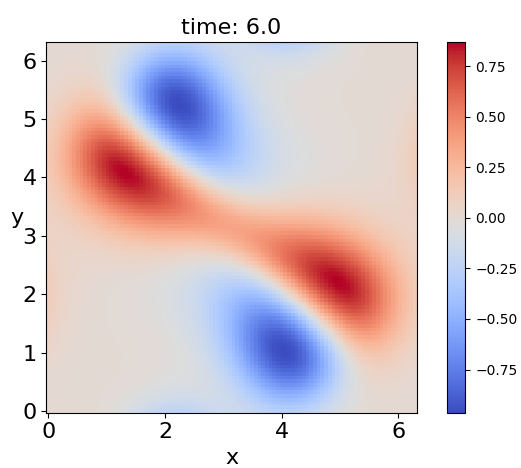
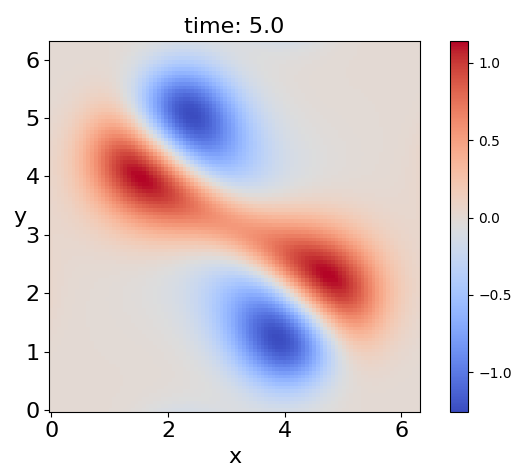
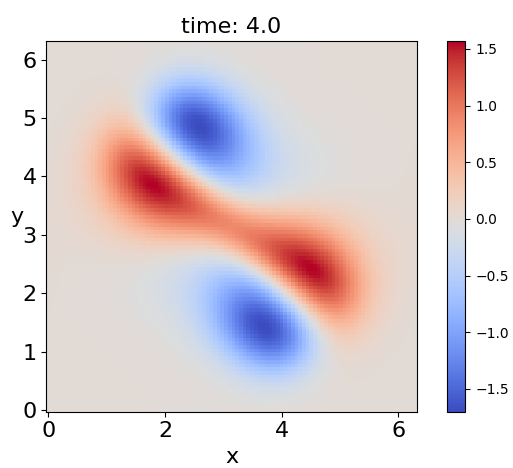
For our example to (5.5), we use $T = 10, L = 2\pi, \nu = 0.5$ and the initial condition for the velocity is,

$$\mathbf{u}_0(x, y) = 10(e^{-5((x-0.5\pi)^2+(y-0.9\pi)^2)} - e^{-5((x-1.5\pi)^2+(y-1.1\pi)^2)})\mathbf{e}_x.$$

We find the initial condition for the coefficients using $\tilde{w}_0 = -ik_x \mathcal{F}\{\mathbf{u}_0\}$. The joint RIDC, pseudo-spectral solution is then acquired by performing the steps above.

The time evolution of the acquired solution to PDE (5.5) is given by the following images in the figure below.





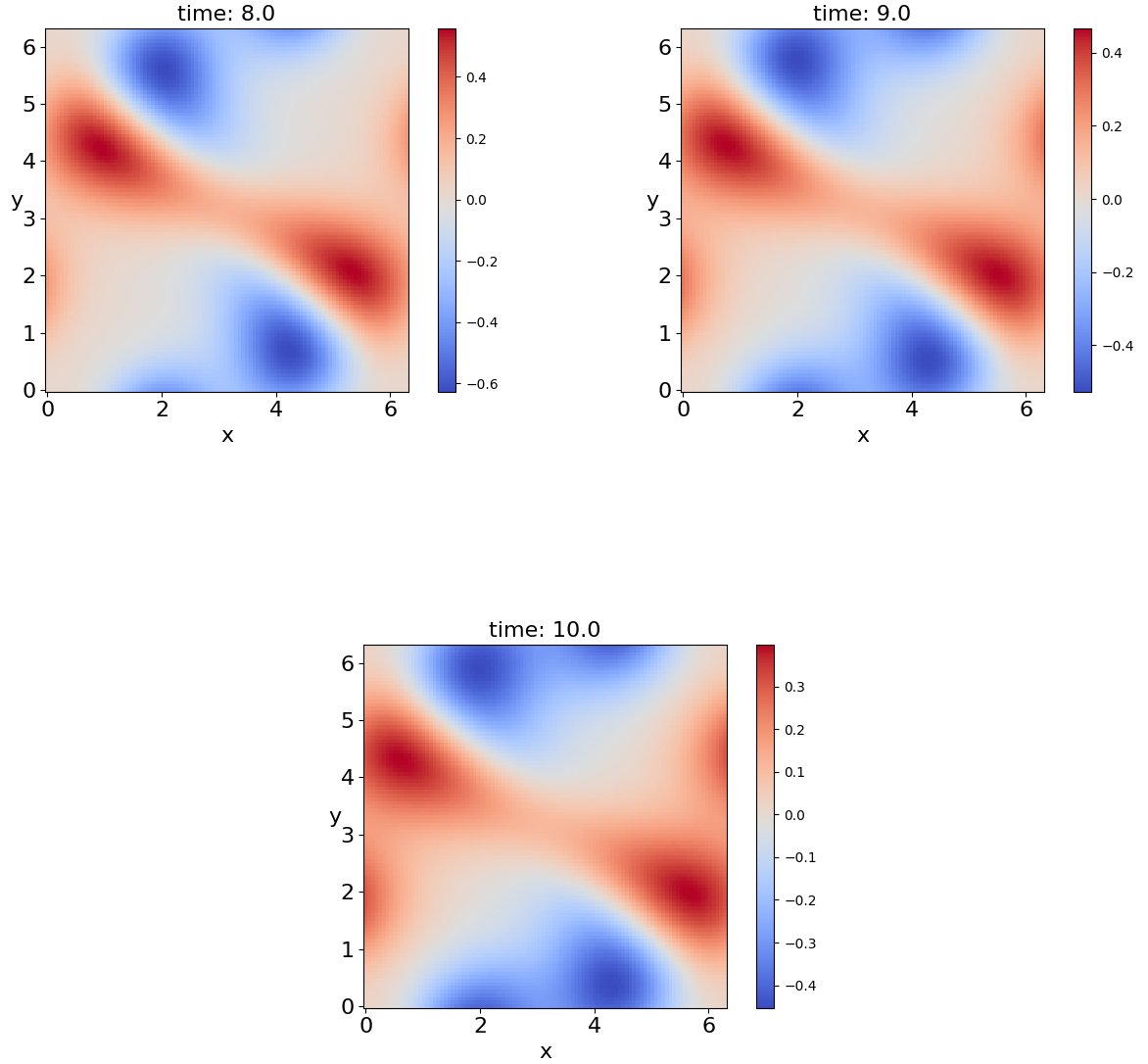


Figure 5.2: The time evolution of the numerical solution to (5.5) using RIDC time integration and the pseudo-spectral method. The parameters used for the integration methods are: $N = 100$, $P = 1$, $J = 10$, $K = 2000$, $M = 3$. We therefore expect this solution to have an order of accuracy in time of 4 and in space of 100.

Chapter 6

Conclusion

To summarise, this report has provided a detailed discussion on the Revisionist Integral Deferred Correction (RIDC) method as an adaptation to the Integral Deferred Correction (IDC) method, with a focus on its parallel implementation. We began by introducing classical time integrators and their properties, followed by an explanation of the IDC and RIDC methodology.

Through empirical analysis, we have demonstrated that the order of accuracy for both methods is given by $P(M+1)$, where P is the order of the underlying time integration method and M is the number of correction levels. We have also shown that RIDC, when combined with the spectral method, can be used to obtain high order and time-efficient solutions to partial differential equations.

We have also emphasised the advantages of parallelism, particularly when dealing with high order time integrators that involve multiple correction levels. With the parallel implementation of RIDC, adding more correction levels does not necessarily increase the computation time since multiple processors can act simultaneously over the staggered nodes in each correction level. This is made possible through the initial setup phase, which creates a staggered node arrangement in the correction levels.

Overall, this report highlights that the RIDC is a highly efficient and accurate method for time integration. Further research in this area can lead to the development of more accurate and efficient methods for solving complex numerical simulations.

Bibliography

- [1] E. O. Brigham and R. E. Morrow. “The fast Fourier transform”. In: *IEEE Spectrum* 4.12 (1967), pp. 63–70. DOI: 10.1109/MSPEC.1967.5217220.
- [2] John Charles Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.
- [3] Rohit Chandra et al. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
- [4] Andrew Christlieb et al. “Revisionist integral deferred correction with adaptive step-size control”. In: *Communications in Applied Mathematics and Computational Science* 10.1 (Mar. 2015), pp. 1–25. DOI: 10.2140/camcos.2015.10.1. URL: <https://doi.org/10.2140%2Fcamcos.2015.10.1>.
- [5] Alok Dutt, Leslie Greengard, and Vladimir Rokhlin. “Spectral deferred correction methods for ordinary differential equations”. In: *BIT Numerical Mathematics* 40 (2000), pp. 241–266.
- [6] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [7] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.
- [8] William H Press et al. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.
- [9] Lloyd N Trefethen. *Spectral methods in MATLAB*. SIAM, 2000.

Appendix A

Fortran Code for Parallel RIDC

```
1  program RIDCEuler
2
3  use omp_lib
4
5  implicit none
6
7  ! Computation time variables
8  double precision :: start_time, end_time, elapsed_time
9
10 ! Function input variables
11 real :: T=1.0
12 complex :: u_0(1)=[1.0]
13 integer :: J=10, K=1E7, M=9
14
15 ! Allocatable function variables
16 double complex, allocatable :: u_hat(:, :), u_init(:, :),
    u_first(:, :, :), f_int(:)
17 double complex, allocatable :: u_new(:, :), u_prev(:, :, :),
    u_corr(:)
18 double precision, allocatable :: S(:, :), t_nodes(:), coef(:)
    , quad_nodes(:)
19 integer, allocatable :: powers(:), int_indices(:)
20
21 ! Function variables
22 double precision :: h, t_curr
23 integer :: N, d, offset, thread_id
24
```

```

25  ! Loop variables
26  integer :: i_do, j_do, m_do, k_do
27
28  ! Print max number of threads
29  call omp_set_num_threads(M+1)
30  print *, "Num Threads: ", omp_get_max_threads()
31
32  ! Initialise timer
33  start_time = omp_get_wtime()
34
35  ! Body of RIDCEuler
36  N = J*(K-1)+1
37  h = T / (N-1)
38  d = size(u_0)
39
40  ! -----
41  ! Calculate quadrature weights matrix
42  allocate(S(M, M+1))
43  allocate(t_nodes(M+1))
44  allocate(powers(M+1))
45
46  do i_do = 1, M+1
47      t_nodes(i_do) = (i_do-1)*h
48      powers(i_do) = M+2-i_do
49  end do
50
51  do i_do = 1, M+1
52      allocate(coef(M+1))
53      coef(1:M) = 0.0
54      coef(M+1) = 1.0
55
56      do j_do = 1, M+1
57          if (j_do /= i_do) then
58              coef = cshift(coef, shift=1) / (t_nodes(i_do)-
59              t_nodes(j_do)) &
60              - coef * t_nodes(j_do) / (t_nodes(i_do)-
61              t_nodes(j_do))
62          end if
63      end do
64
65      do j_do = 1, M
66          S(j_do, i_do) = sum(coef * t_nodes(j_do+1)**powers /
67          powers) &
68          - sum(coef * t_nodes(j_do)**powers / powers)
69      end do
70      deallocate(coef)

```



```

68 end do
69
70 deallocate(t_nodes)
71 deallocate(powers)
72
73 ! -----
74 ! Start of RIDC Method
75
76 ! Initialise initial condition
77 allocate(u_init(d))
78 u_init = u_0
79
80 ! Initialise numerical solution array
81 allocate(u_hat(N, d))
82 u_hat(1, :) = u_init
83
84 ! Loop over the J time intervals
85 do j_do = 1, J
86     print *, "Time Interval:", j_do
87
88     ! -----
89     ! Setup Phase: Sequentially compute only the nodes
90     ! necessary for each correction level such that the
91     ! last has exactly P(M+1) calculated nodes.
92
93
94     ! Initialise array containing the first 2M-m+1
95     ! vectors for each correction level,
96     ! shape: (corr_lvl, time, vec_dim)
97     allocate(u_first(M+1, 2*M+1, d))
98     u_first(:, 1, :) = spread([u_init], dim=1, ncopies=M+1)
99
100    ! Prediction loop (Euler Method)
101    ! Initialise start time
102    t_curr = (j_do-1)*T/J
103    do i_do = 1, 2*M
104        ! Get prediction value
105        u_first(1, i_do+1, :) = u_first(1, i_do, :) &
106            + h*f(u_first(1, i_do, :), t_curr)
107
108        ! Update time value
109        t_curr = t_curr + h
110
111    end do
112
113    ! Correction loops (Euler Update)

```

```

114     do m_do = 2, M+1
115         ! Initialise start time
116         t_curr = (j_do-1)*T/J
117         do i_do = 1, 2*M+1-m_do
118             allocate(int_indices(M+1))
119             allocate(quad_nodes(M+1))
120
121             ! Get quadrature offset and integration points
122             if (i_do >= M+1) then
123                 offset = M
124                 do k_do = 1, M+1
125                     int_indices(k_do) = k_do+i_do-M
126                 end do
127             else
128                 offset = i_do
129                 do k_do = 1, M+1
130                     int_indices(k_do) = k_do
131                 end do
132             end if
133
134             ! Get quadrature time nodes
135             do k_do = 1, M+1
136                 quad_nodes(k_do) = t_curr+(k_do-offset)*h
137             end do
138
139             allocate(f_int(d))
140
141             ! Get quadrature approximation
142             f_int(:) = 0
143             do k_do = 1, M+1
144                 f_int = f_int + S(offset, k_do) &
145                     *f(u_first(m_do-1, int_indices(k_do), :),
quad_nodes(k_do))
146             end do
147
148             deallocate(int_indices)
149             deallocate(quad_nodes)
150
151             ! Get correction value
152             u_first(m_do, i_do+1, :) = u_first(m_do, i_do, :)
&
153                 + h*(f(u_first(m_do, i_do, :), t_curr) - f(
u_first(m_do-1, i_do, :), t_curr)) &
154                 + f_int
155
156             deallocate(f_int)

```

```

157
158         ! Update time value
159         t_curr = t_curr + h
160
161     end do
162 end do
163
164     ! Put the time intervals first M+1 values for the
165     ! final correction level into the solution array
166     u_hat((j_do-1)*(K-1)+1:(j_do-1)*(K-1)+M+1, :) = u_first(M
+1, 1:M+1, :)
167
168     ! -----
169     ! Parallel Phase: Simultaneously calculate the new
170     ! correction values.
171
172     ! Initialise array for holding the new values
173     allocate(u_new(M+1, d))
174
175     ! Initialise array for holding the M+1 most recent
176     ! values calculated for each correction level
177     ! shape: (corr lvl, time, vec dim)
178     allocate(u_prev(M+1, M+1, d))
179     do m_do = 1, M+1
180         do i_do = 1, M+1
181             u_prev(m_do, i_do, :) = u_first(m_do, i_do+M+1-
m_do, :)
182         end do
183     end do
184
185     deallocate(u_first)
186     allocate(u_corr(d))
187
188     ! Loop over the time step, we stop if i which
189     ! represents the index of the current node is
190     ! >= K+1
191
192     ! This is the section we run in parallel
193     !$omp parallel private(i_do, m_do, k_do, t_curr, u_corr,
quad_nodes, f_int, thread_id)
194     thread_id = omp_get_thread_num()
195
196     do i_do = M+1, K-1
197         ! Prediction loop
198         if (i_do+M < K .and. thread_id == 0) then
199             ! Get current time

```

```

200         t_curr = (j_do-1)*T/J + (i_do+M-1)*h
201         ! Get prediction value
202         u_corr = u_prev(1, M+1, :) + h*f(u_prev(1, M+1,
:), t_curr)
203         ! Add prediction to new values array
204         u_new(1, :) = u_corr
205     end if
206
207     ! Correction loop
208     do m_do = 2, M+1
209         if (i_do+M+1-m_do < K .and. thread_id == m_do-1)
then
210             ! Get current time
211             t_curr = (j_do-1)*T/J + (i_do+M-m_do)*h
212
213             ! Get quadrature time nodes
214             allocate(quad_nodes(M+1))
215
216             do k_do = 1, M+1
217                 quad_nodes(k_do) = t_curr+(k_do-M)*h
218             end do
219
220             ! Get quadrature approximation
221             allocate(f_int(d))
222             f_int(:) = 0
223             do k_do = 1, M+1
224                 f_int = f_int + S(M, k_do) &
225                     *f(u_prev(m_do-1, k_do, :),
quad_nodes(k_do))
226             end do
227
228             deallocate(quad_nodes)
229
230             ! Get correction value
231             u_corr = u_prev(m_do, M+1, :) &
232                 + h*(f(u_prev(m_do, M+1, :), t_curr) - f(
u_prev(m_do-1, M, :), t_curr)) &
233                 + f_int
234
235             deallocate(f_int)
236
237             ! Add correction value to new values array
238             u_new(m_do, :) = u_corr
239
240         end if
241

```

```

242         if (m_do == M+1 .and. thread_id == M) then
243             ! Put the final correction value into
solution array
244             u_hat((j_do-1)*(K-1)+i_do+1, :) = u_corr
245         end if
246
247     end do
248
249     !$omp barrier
250
251     if (thread_id == 0) then
252         ! Update the previous values array
253         u_prev(:, 1:M, :) = u_prev(:, 2:M+1, :)
254         u_prev(:, M+1, :) = u_new
255     end if
256
257 end do
258
259 !$omp end parallel
260
261
262 !
-----
263
264 ! End Phase: Update the initial condition for the next
time
265 ! interval. Also deallocate arrays
266
267 ! Drop down final correction value to lower correction
levels
268 u_init = u_prev(M+1, M+1, :)
269
270 deallocate(u_new)
271 deallocate(u_prev)
272 deallocate(u_corr)
273
274 end do
275
276 ! Get elapsed time
277 end_time = omp_get_wtime()
278 elapsed_time = end_time - start_time
279
280 print *, "Elapsed Time: ", elapsed_time
281 read(*, *)
282 contains

```

```

283
284 ! RHS of IVP,  $u'(t)=f(u,t)$ 
285 function f(u, t)
286     implicit none
287
288     double complex, intent (in) :: u(1)
289     double precision, intent (in) :: t
290
291     double complex :: f(1)
292
293     f = 2.0*t*u
294 end function f
295
296 end program RIDCEuler

```

