Numerical Solution to the Initial Value Problem by a Predictor-Corrector Method: Applications to n-Body Problem and Pendula

Joel Biffin

April 25, 2019

Contents

Abstract

Initial Value Problems primarily arise from the modelling of natural phenomena where often we are able to desribe some quantity's rate of change as a function of itself coupled with time. This report will focus on Initial Value Problems arising from models using Ordinary Differential Equations. It is notable that many of the breakthroughs in the field of solving Partial Differential Equations have arisen from spacially discretising the problem and approximating solutions to the discretised Initial Value Problem.

Leibniz and Newton are often cited as the fathers of the field of differential equations. Similar to Newton, in this report we will be considering quantities which change with respect to time, but all of the methods and analyses apply for any other variable too.

Until the late 19th century, numerical algorithms to approximate solutions to Initial Value Problems could not usefully be applied. Since then contributions from Adams, Milne, Butcher and Gear (to name a few) have been used in computations an unimaginable number of times as a result of now readily available computing power.

In this report, established methods will be analysed and reviewed comparatively in terms of implementation complexity, computational cost and accuracy.

Chapter 1

Background

1.1 Notation

Throughout this report, vector quantities and functions will be written in lowercase bold type ($\mathbf{u}, \mathbf{f}$), scalar quantities and functions will be written in lowercase type (t, f), and iterative schemes will use subscripting to represent the i-th iteration ($u_i, \mathbf{u}_i$) of a method. Generally, vector quantities will be expressed as such rather than element-wise, unless otherwise stated.

Chapter 3 gives reference to *solvers*–this italicised

1.2 Initial Value Problems

An Initial Value Problem (IVP) can generally be written in the form,

$$\begin{aligned} \frac{d\mathbf{u}(t)}{dt} &= \mathbf{f}(t, \mathbf{u}), \quad t \in [a, b] \\ \mathbf{u}(a) &= \mathbf{u}_0, \quad \text{(given)}, \end{aligned} \tag{1.1}$$

using vector quantities. It is worth noting that this is in the form of a 1st order system of differential equations, this is due to the fact we can reduce any n-th order equation to a system of first order ones (see section).

1.3 Where Do They Come From?

As discussed in the abstract, IVPs often arise from, but are not restricted to, the modelling of natural phenomena.

Newton's 2nd Law of Motion describes the proportionality of the rate of change of velocity to the resultant force acting on a body. This coupled with an initial condition $\mathbf{v}(a) = \mathbf{v}_0$, gives an IVP. In electrical circuit modelling, the flow of charge q from a capacitor of known initial charge $q(0) = q_0$ can too be modelled by an IVP in the form of (1.1).

1.4 Existence & Uniqueness of Solutions

Before discussing methods used to approximate solutions to IVPs, it is important to ascertain whether solutions do in fact exist and, if so, whether or not they are unique. We will use Lipschitz continuity to do so.

Theorem 1.1. *Let $f(t, u)$ be a continuous function for all u and $t \in [a, b]$, and suppose that f satisfies the Lipschitz condition,*

$$|f(t, v) - f(t, w)| \leq L|v - w|, \quad \forall v, w \text{ and } t \in [a, b]. \tag{1.2}$$

Then, for any u_0 there exists a unique continuously differentiable function $u(t)$ defined on $[a, b]$ satisfying (1.1).

This result is incredibly important when modelling environments and phenomena. If a model equation does not satisfy our Lipschitz condition this is a real indicator of whether or not our model correctly represents our system.

For the remainder of this report we will be concerned with problems which do satisfy our Lipschitz condition (1.2).

1.5 Analytical Methods vs Numerical Methods

Analytical methods can be used to solve IVPs but only for a very limited number of special cases. In terms of computing, there are a number of software packages that are able to solve these differential equations symbolically (for example, MATLAB's 'MuPAD'). Since it is the case that the majority of ordinary differential equations cannot by solved analytically, these algorithms are of little use (and it's worth noting that these algorithms are notoriously costly).

This inability to analytically find solutions led to the discussion of numerical methods to approximate solutions to IVPs at a specific t. The general approach for these numerical methods begins with discretising our independent variable t into a mesh. This mesh is an increasing monotonic sequence of time values $\{t_i\}_{i=0}^{n}$, where $[a, b]$ has been split into n (not necessarily equally spaced) intervals such that $a = t_0$ and $b = t_n$.

In practice, numerical methods are the best approach to finding solutions to IVPs and when used appropriately they produce incredibly useful results to equations that we would otherwise be unable to solve.

In the next chapter we will be discussing what it is that makes a numerical algorithm appropriate for a given problem. We will also be investigating the accuracy of our approximations.

Chapter 2

Numerical Methods

Algorithms which approximate solutions to (1.1) fall into three categories: Taylor Series methods, Linear Multistep Methods (LMMs) and Runge-Kutta methods. It can be viewed that all three of these families are generalisations of Euler's Method (2.1.1).

The focus of this project primarily lies with Linear Multistep Methods and using them in the form of a predictor-corrector pair. Runge-Kutta methods are too used and evaluated here, but are only featured in case studies as starting schemes for LMMs.

For the purposes of software design we have classified Euler's Method as a One-Step method, and similarly Runge-Kutta methods are classified as One-Step methods too. Unless explicitly stated, all methods will use a fixed step-size $h = \frac{b-a}{n}$ (where n is the number of time intervals) producing an equally spaced mesh.

2.1 One-Step Methods

2.1.1 (Forward) Euler's Method

Euler's method is given by the iterative scheme,

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\mathbf{f}(t_i, \mathbf{u}_i). \tag{2.1}$$

This scheme is *explicit* since we can calculate $\mathbf{u}_{i+1}$ using a combination of $\mathbf{u}_i$ and t_i which are known at the start of an iteration.

Definition 2.1. An iterative scheme is called **explicit** if at the beginning of an iteration we can express $\mathbf{u}_{i+1}$ in terms of values which have already been computed or can be computed by simple substitution during the iteration. In contrast, **implicit** schemes have $\mathbf{u}_{i+1}$ appearing on both 'sides' of our scheme, so an algebraic equation must be solved at each iteration to compute $\mathbf{u}_{i+1}$.

We will discuss later on how the explicit nature of a method impacts its stability.

2.1.2 (Backward) Euler's Method

This method can be viewed as an *implicit* variant of (2.1), and is given by

$$\mathbf{u}_{i+1} = \mathbf{u}_i + h\mathbf{f}(t_{i+1}, \mathbf{u}_{i+1}). \tag{2.2}$$

Since $\mathbf{u}_{i+1}$ appears on both sides of our expression, we must solve

$$\mathbf{g}(\mathbf{u}_{i+1}) = \mathbf{u}_{i+1} - \mathbf{u}_i - h\mathbf{f}(t_{i+1}, \mathbf{u}_{i+1}) = \mathbf{0} \tag{2.3}$$

for $\mathbf{u}_{i+1}$ at every iteration.

Of course, we could solve this symbolically at a cost, but in practice we use any numerical (vector) root finding algorithm (Secant Method, Newton's Method etc.).

Originally in this project Newton's Method was applied using the `scipy.optimize.newton` package in Python where the Jacobian matrix was approximated via Finite Differences. As is conventional, the initial 'guess' solution was computed via (2.1). Unfortunately this method led to some non-convergent solutions being returned. Eventually the decision was made to use the `scipy.optimize.fsolve` package which implements a variant of the Powell hybrid method.

2.2 Global & Local Truncation Error

As with any numerical approximations, our methods will introduce errors. Here we will include floating point errors in our truncation error considerations, so floating point errors are not noted directly but are considered in implementation.

Definition 2.2. The **Global Truncation Error** (GTE), denoted $e_i(t)$, is the total error in our approximation. It is given by $e_i = u(t_i) - u_i$, where i represents the i-th iteration in the standard way (see Subsection 1.1).

Definition 2.3. The **Local Truncation Error** (LTE), denoted $\tau(h)$, for a numerical method is the error introduced in one single iteration. This is the error introduced in computing $\mathbf{u}_{i+1}$ when all previously computed values are taken to be exact.

GTE is really just a consequence of LTE and due to this it is helpful to address our full attention to the analysis of LTE.

For any method we can derive an expression for the order of LTE by using a Taylor Series expansion for the exact solution to the ODE and take the difference between that and our iterative scheme (when all previous values are taken to be exact). Doing this for Euler's method is shown:

Suppose $u_i = u(t_i)$, then Euler's method gives us,

$$\begin{aligned} u_{i+1} &= u(t_i) + hf(t_i, u(t_i)) \\ &= u(t_i) + hu'(t_i), \quad \text{since } u'(t_i) = f(t_i, u(t_i)). \end{aligned} \tag{2.4}$$

Our Taylor expansion for our exact solution is,

$$u(t_{i+1}) = u(t_i + h) = u(t_i) + hu'(t_i) + \frac{h^2}{2}u''(\xi_i) + O(h^3), \quad \xi_i \in (t_i, t_{i+1}). \tag{2.5}$$

Taking the difference of these two expressions leaves us the remainder,

$$\tau(h) = u(t_{i+1}) - u_{i+1} = \frac{h^2}{2}u''(\xi_i) + O(h^3) = O(h^2), \tag{2.6}$$

so we know that for Forward Euler we have $\tau(h) = O(h^2)$.

It's worth noting that a similar process can be carried out to find an expression for GTE but without the assumption that $u_i = u(t_i)$.

Definition 2.4. A method is said to be of **order** p if its LTE satisfies $\tau(h) = O(h^{p+1})$. If $p \geq 1$ then we say that the method is consistent of order p.

With this definition we know that Forward Euler is an order 1 method – so too is Backward Euler. In addition, when our method is order p, we know that the GTE, $e_i = O(h^p)$ too.

2.3 Convergence & Stability

Efforts in using these numerical algorithms would be fruitless if the results obtained were not representative of the *true* solution. So it is important that we know a method is *convergent* before using it.

Definition 2.5. A method is said to be **0-stable** if a small perturbation $\boldsymbol{\varepsilon}$ ($\|\boldsymbol{\varepsilon}\| << 1$) to $\mathbf{u}_0$ causes the numerical solution to change at most by $K\boldsymbol{\varepsilon}$, where K is independent of h.

We will later discuss the *root condition* which can be used as a quantitative method to establish 0-stability, but for the moment this qualitative definition is all we need.

Theorem 2.1 (Dahlquist's Equivalence Theorem)**.** *A numerical method is said to be convergent of order p if and only if it is 0-stable and consistent of order p.*

Corollary 2.1.1. A convergent numerical method has the property that,

$$\lim_{h\to 0}\|\mathbf{u}(t_i) - \mathbf{u}_i\| = 0. \tag{2.7}$$

In practice, we are computing with floating point numbers so, in an arbitrary precision, by taking h to be excessively small then the accuracy that we might theoretically gain will be counteracted by round-off errors.

2.3.1 Absolute Stability

Given that a method is *0-stable* we know that if we take h to be sufficiently small we acheive a very good representation of the true solution. But how small does h have to be? Are there any limitations on how *large* h is?

Consider the scalar *test equation*,

$$u' = \lambda u, \quad \lambda \in \mathbb{C}, \tag{2.8}$$

whose solution is known, $u(t) = ce^{\lambda t}$ for some constant c. When $\mathbb{R}\text{e}\{\lambda\} < 0$, our solution converges to 0 as $t \to \infty$. This results in the absolutely decreasing sequence of approximations $\{u_i\}_{i=0}^{\infty}$.

We want our numerical methods to share this property when acting on the test equation.

Definition 2.6. The **region of absolute stability** is the set of $h\lambda \in \mathbb{C}$ where $\{u_i\}_{i=0}^{\infty}$ is an absolutely decreasing sequence.

We can find the region of absolute stability by applying our numerical method to the test equation. For forward Euler we attain the region $|1+h\lambda| \leq 1$ which provides us a very restrictive upper bound on the value of h we can choose. Applying backward Euler gives the region $|1-h\lambda|^{-1} \leq 1$, which is satisfied for all values of h, λ where $\mathbb{R}\text{e}\{\lambda\} \leq 0$. Hence the entire left half-plane is in the region of stability and there is no restriction on the value of h we can choose.

Definition 2.7. A numerical method having the full left-half plane in its region of absolute stability is said to be **A-stable**.

2.3.2 Stiffness

Ideally we would be able to choose our step size h based solely on the accuracy we want from our computation. Unfortunately this is not always the case – as highlighted by our region of absolute stability for forward Euler. The *stiffness* of a system helps to characterise this problem.

Definition 2.8. An IVP is said to be **stiff** if the step size required to satisfy our absolute stability requirement is significantly smaller than the step size required to approximate at some desired accuracy.

Example 2.1. Consider the IVP,

$$\begin{cases} u'(t) = 100u(t), & \text{on } t \in [0, 0.2], \\ u(0) = 10. \end{cases} \tag{2.9}$$

Figure 2.1 demonstrates the impact of our equation's stiffness on our results using forward Euler. By method of substitution, our absolute stability expression tells us that in order to compute stable approximations using (2.1) we must use $h \leq 0.02$. In contrast, we have no bound on our choice of h in terms of stability for backward Euler. This is highlighted by the general shape of our graphs in comparison to results from forward Euler.

In summary, we must be careful when choosing numerical methods to solve stiff equations. Depending on the stiffness, we must decide between the performance of choosing an explicit method with smaller steps or using an implicit method with larger steps, but needing to numerically solve an equation such as (2.3) at each iteration.

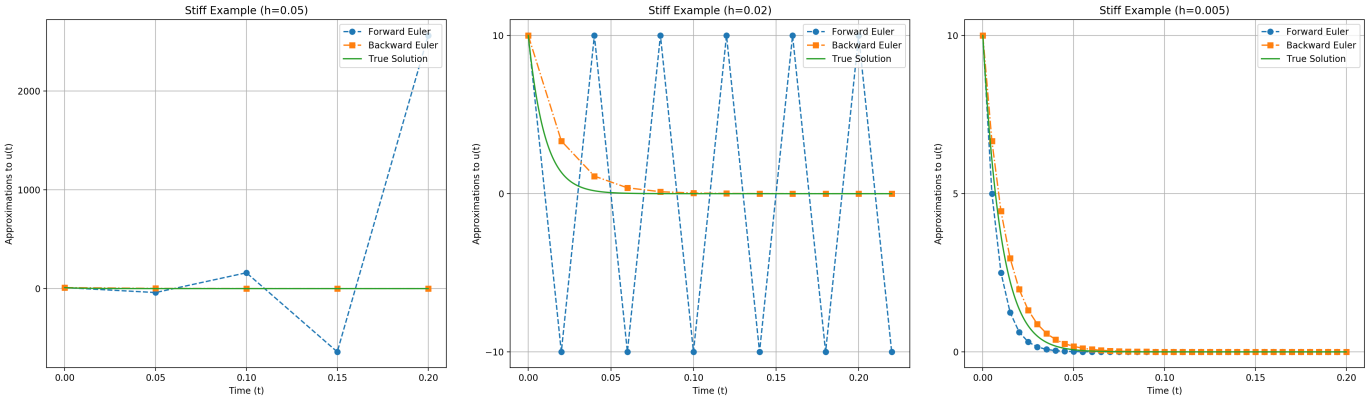

Figure 2.1: Demonstration of stiffness using (2.1) & (2.2) with $h = 0.05, 0.02, 0.005$ on (2.9)

2.4 Runge-Kutta Methods

Runge-Kutta methods (first discussed by Runge 1895) look to solve the problem of *improving the accuracy of our approximations without the need to take significantly smaller time intervals?*. Here we won't discuss Runge-Kutta methods in much depth as it is outside of the scope of this project, but we will mention asymptotic complexity and derivations.

From attempting to integrate (1.1), we can apply a Gauss quadrature rule,

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \int_{t_i}^{t_{i+1}} \mathbf{u}'(t)dt, \tag{2.10}$$

to solve our differential equation. It is with this idea that Runge-Kutta methods were established.

The general notion is that if we can evaluate intermediate values (or *stages*) between $\mathbf{u}_i$ and $\mathbf{u}_{i+1}$, we can better approximate $\mathbf{u}_{i+1}$.

2.4.1 4-Stage Runge-Kutta

Generally the most popular Runge-Kutta method is known as the *Classical Runge-Kutta* which is in fact a special case of the generalised 4-stage Runge-Kutta. The generalised form has 10 free parameters, optimising these parameters to yield the highest possible *order* gives us our 'Classical' method. This method is given by the iterative scheme,

$$\begin{aligned}
\mathbf{u}_{i+1} &= \mathbf{u}_i + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \quad \text{where} \\
\mathbf{k}_1 &= \mathbf{f}(t_i, \mathbf{u}_i), \\
\mathbf{k}_2 &= \mathbf{f}(t_i + \frac{h}{2}, u_i + \frac{1}{2}\mathbf{k}_1), \\
\mathbf{k}_3 &= \mathbf{f}(t_i + \frac{h}{2}, \mathbf{u}_i + \frac{1}{2}\mathbf{k}_2), \\
\mathbf{k}_4 &= \mathbf{f}(t_i + h, \mathbf{u}_i + \mathbf{k}_3).
\end{aligned} \tag{2.11}$$

This method is fourth order as $\tau(h) = O(h^5)$.

In practice, Runge-Kutta methods have great accuracy properties and for small examples often yield sufficient results at a reasonable time cost. Their downsides however, are in the fact that every iteration we are computing a number of intermediate values and then discarding them at each step. These intermediate function evaluations actually dominate our algorithm's running time. Consequently, when working on large systems of complicated ODEs our computations become very inefficient.

2.5 Linear Multistep Methods

In a similar vein to Runge-Kutta methods, Linear Multistep Methods look to use more information from $t < t_{i+1}$ to improve our approximations. In contrast to Runge-Kutta methods, LMMs use previously computed values in their schemes which helps to avoid the problem highlighted at the end of Section 2.4.

Unlike aforementioned one-step methods, LMMs are not *self-starting* and in order to carry out the first steps, a starting scheme must be decided on. Often a Runge-Kutta method of appropriate order is used as a starting routine. This method is chosen to have an order equal to or higher than the coupled LMM, to preserve asymptotic accuracy.

2.5.1 Adams Methods

Adams methods were devised in the late 19th century (prior to Runge). They are derived from trying to solve the integral equation,

$$\mathbf{u}(t_{i+1}) = \mathbf{u}(t_i) + \int_{t_i}^{t_{i+1}} \mathbf{f}(t, \mathbf{u}(t))dt, \tag{2.12}$$

by approximating our integrand by a polynomial which interpolates our previously computed $\mathbf{u}_j$ values.

A k-step *Adams-Bashforth* method is an *explicit* Adams scheme which interpolates to k values to approximate our integrand, $\mathbf{f}(t, \mathbf{u}(t))$. A k-step method has LTE of $\tau(h) = O(h^{k+1})$, so is order k. It's worth noting that over the bounds of our integral, (t_i, t_{i+1}), we are *extrapolating* which is often unreliable.

The 2 and 3-step Adams-Bashforth methods are given by their iterative schemes,

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}(3\mathbf{f}(t_i, \mathbf{u}_i) - \mathbf{f}(t_{i-1}, \mathbf{u}_{i-1})), \tag{2.13}$$

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{12}(23\mathbf{f}(t_i, \mathbf{u}_i) - 16\mathbf{f}(t_{i-1}, \mathbf{u}_{i-1}) + 5\mathbf{f}(t_{i-2}, \mathbf{u}_{i-2})). \tag{2.14}$$

Implicit Adams schemes are known as *Adams-Moulton* methods. They are derived in a similar way to Adams-Bashforth methods except a k-step *Adams-Moulton* method looks to approximate the integrand with a polynomial which interpolates to $k+1$ points. This results in integrating over an interpolated region which is much more reliable than the explicit schemes.

The trade-off for this increase in stability is that we must solve a non-linear equation (similar to (2.3)) at every iteration – which we know is costly.

The 1 and 2-step Adams-Moulton methods are given by their iterative schemes,

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{2}(\mathbf{f}(t_i, \mathbf{u}_i) + \mathbf{f}(t_{i+1}, \mathbf{u}_{i+1})), \tag{2.15}$$

$$\mathbf{u}_{i+1} = \mathbf{u}_i + \frac{h}{12}(5\mathbf{f}(t_{i+1}, \mathbf{u}_{i+1}) + 8\mathbf{f}(t_i, \mathbf{u}_i) - \mathbf{f}(t_{i-1}, \mathbf{u}_{i-1})). \tag{2.16}$$

A k-step Adams-Moulton method is order $k+1$.

2.5.2 Nyström Methods

Similar to Adams, Nyström used the fundamental theorem of calculus to derive an integral equation which could be solved by approximating the intergrand. Unlike in (2.12), Nyström's integral equation acts over the larger time interval (t_{i-1}, t_{i+1}). The integral equation is,

$$\mathbf{u}(t_{i+1}) = \mathbf{u}(t_{i-1}) + \int_{t_{i-1}}^{t_{i+1}} \mathbf{f}(t, \mathbf{u}(t))dt. \tag{2.17}$$

Nyström methods are *explicit*. In this report we will apply the 2-step *Leapfrog method*, which is defined by the iterative scheme,

$$\mathbf{u}_{i+1} = \mathbf{u}_{i-1} + 2h\mathbf{f}(t_i, \mathbf{u}_i), \tag{2.18}$$

and has order 2.

2.5.3 Backwards Differentiation Formulae

Unlike Nyström and Adams methods, *Backwards Differentiation Forumlae* (BDF) methods do not look to numerical integration for their derivations. Instead, the idea is to start with an interpolating polynomial to directly approximate our solution function. Then, by numerically differentiating our interpolant, we substitute this approximation into our IVP and derive our iterative scheme.

BDF methods are *implicit* methods, so they come with the same caveats as the Adams-Moulton methods – that is, the trade off between stability and computational cost. BDF methods are *A-stable*, so are appropriate to solve *stiff* equations (popularised by Gear, 1971).

The 2-step BDF method (or *BDF-2*) is given by the iterative scheme,

$$\mathbf{u}_{i+1} = \frac{4}{3}\mathbf{u}_i - \frac{1}{3}\mathbf{u}_{i-1} + \frac{2h}{3}\mathbf{f}(t_{i+1}, \mathbf{u}_{i+1}), \tag{2.19}$$

and is second order.

2.6 Predictor-Corrector Methods

Ignoring their computational cost, *implicit* methods are more favourable when wanting a one-size-fits-all family of algorithms due to their excellent stability properties. But how do we reduce their cost?

Functional iteration is the natural way that one might attempt to evaluate $\mathbf{u}_{i+1}$ at each step. But this actually creates more problems than it solves (especially in stiff equations) and only converges to our solution *linearly*. We have already mentioned the use of numerical root finders. Newton's method is a popular choice as on paper it converges quadratically – yielding a faster convergence for a similar cost. In fact this is not the case. As discussed in [8], when the Jacobian matrix is approximated via finite differences (as it is in most ODE solvers), Newton's method converges *superlinearly* with an order of convergence $\sqrt{2}$. Predictor-corrector methods avoid these problems and attempt to make good use of the desirable properties of both *explicit* and *implicit* methods.

Definition 2.9. A **predictor-corrector** method is a method pairing with an *explicit* predictor to produce a value $\mathbf{u}^*_{i+1}$ which is substituted into an *implicit* corrector to produce our desired value, $\mathbf{u}_{i+1}$. Predictor-correctors have the form,

$$\mathbf{u}^*_{i+1} = \mathbf{\Phi}^*(\text{ previously computed values of } \mathbf{u}), \tag{2.20}$$

$$\mathbf{u}_{i+1} = \mathbf{\Phi}(\mathbf{u}^*_{i+1}, \text{previously computed values of } \mathbf{u}), \tag{2.21}$$

where $\mathbf{\Phi}^*, \mathbf{\Phi}$ are chosen explicit and implicit schemes respectively.

There are different variants of predictor-correctors, PEC, PECE and P(EC)^{k}E. These letters refer to the steps carried out in implementation – *Predict, Evaluate (derivative), Correct, Evaluate (derivative)*. For the purposes of this project, all predictor-correctors followed the PECE structure to simplify implementation.

It is common for Adams-Bashforth and Adams-Moulton methods to be used as a predictor corrector pair. We will also be considering the Forward-Backward Euler, Heun and Leapfrog-BDF2 pairings.

2.7 Adaptive Step-Size

Due to Milne, predictor-corrector methods give us an inexpensive way to approximate our LTE at each iteration. This error estimate is known as *Milne's device* and is given by,

$$\text{LTE} \approx \frac{C_{p+1}}{C_{p+1} - C^*_{p+1}}(\mathbf{u_{i+1}} - \mathbf{u^*_{i+1}}), \tag{2.22}$$

where p is the order of our scheme and C_{p+1}, C^*_{p+1} are the error constants for the predictor and corrector respectively).

This error estimate allows us to optimise our computations so that when our LTE is high we can *decrease* our step-size to reduce our error and *increase* our step size when our LTE is low. We will be adapting our step-size using the formula,

$$h_{i+1} = h_i \left(\frac{\text{tol}}{\|\text{LTE}\|_2} \right)^{\frac{1}{p+1}}, \tag{2.23}$$

where tol is some user defined tolerance.

When using one-step predictor-corrector pairs, we can go right ahead and apply (2.23). Unfortunately, the same cannot be said for LMMs, since in deriving their iterative schemes we have assumed a fixed h. This leaves two options,

(i) derive new schemes which account for adaptive h, or

(ii) interpolate backwards to evaluate values of $\mathbf{u}$ using this step's fixed h.

In our implementation, we used (i) by deriving adaptive step schemes using Lagrange interpolation (see Appendix ??).

Chapter 3

Design & Implementation

3.1 Choice of Python

The numerical methods discussed in Chapter 2 were implemented in Python primarily due to its *NumPy* package and extensive *Matplotlib* graphing capabilities.

NumPy provides many optimised vector operations in the box which overload standard binary operations. This allows for flexible code which need not distinguish between vector or scalar operations, helping to maintain a higher-lever approach when programming.

As mentioned in 2.1.2, we use another python package *SciPy* for its modified Powell hybrid method to solve our non-linear equations in implicit methods.

Clearly, our algorithms would run faster if they were implemented in a lower level language such as C/C++, but the focus of the project was to investigate the different methods rather than implementing them in the most performant way.

3.2 Object-Oriented Design

Since our methods have most of the same requirements as each other, an object-oriented approach was chosen to maximise code reuse. In this design, we give each method their own class which is itself the child of some *abstract* class which inherits directly from the *abstract Solver* class.

Figure 3.1 shows a reduced representation of the inheritance structure of our different *Solver* classes (where the `ForwardEulerSolver` and `LeapfrogSolver` are just examples of `OneStepSolver` and `MultiStepSolver`'s children).

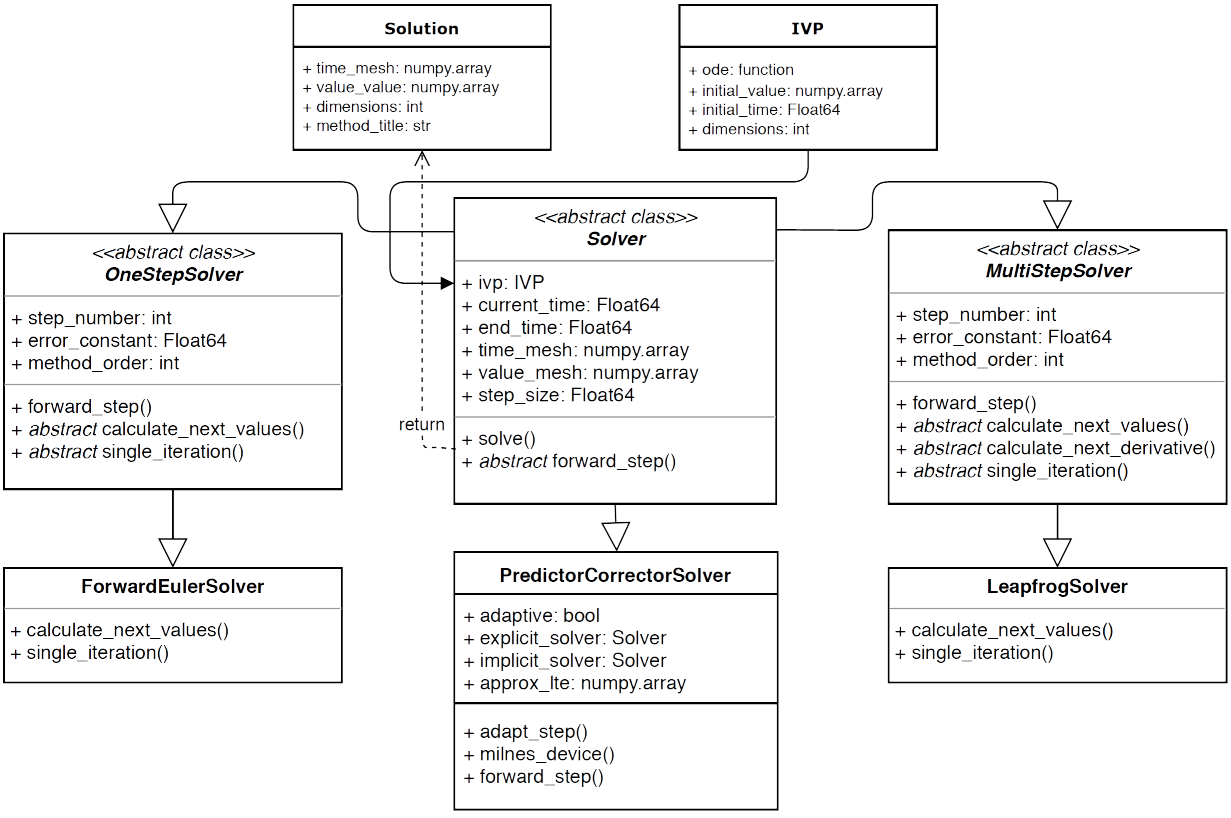

Figure 3.1: UML diagram of Solver class heirarchy

Following Section 2, we categorised our numerical methods into one-step and multi-step methods, which appeared a natural way to split our class definitions. The major difference between `OneStepSolver` and `MultiStepSolver` is that in the latter class we cache our function evaluations between iterations.

Aside from predictor-corrector pairs, when comparing two methods we see that they semantically only differ in their iterative scheme (implemented in the `calculate_next_values` and `single_iteration` methods). Therefore all methods share the same `solve` which is inherited from our `Solver` class.

```
1  def solve(self):
2      while self.current_time <= self.end_time:
3          self.forward_step(step_counter)
4
5      self.solution = Solution(self.time_mesh, self.value_mesh)
```

Listing 1: Outline of `solve` method for a *Solver*

Since our *Solver* classes have numerous 'helper' instance variables the decision was made to produce a (lighter weight) `Solution` object which contains only the matrix of results and time mesh. This allowed for a simpler reporting process when wanting to graph results.

`OneStepSolver` and `MultiStepSolver` have `forward_step` methods of the form,

```
1  def forward_step(self, step):
2      # next_step_size is class property, so it can vary between steps
3      this_h = self.next_step_size
4      self.time_mesh[step] = self.time_mesh[step - 1] + this_h
5      self.value_mesh[step] = self.calculate_next_values(step, this_h)
6      self.derivative_mesh[step] = self.calculate_next_derivative(step)
```

Listing 2: Outline of `forward_step` method for a *Solver*

where the `OneStepMethod` class omits line 6.

Since Multi-step methods are not self starting, `MultiStepSolver` objects are coupled with a `OneStepSolver` object when they are instantiated. If otherwise unspecified, the *Classical Runge-Kutta* method (2.11) is used.

3.2.1 Predictor-Corrector Implementation

Prior to the implementation of the predictor-corrector, standalone algorithms operated on the basis of calling their `solve` method which, in turn, iteratively calls instance methods, mutating the state of our *solver* object. Calling methods to update instance variables is set out by our `Solver` class.

Given that predictor-correctors are themselves *solvers*, our class representation should reflect this. So `PredictorCorrectorSolver` is a child class of `Solver` (as shown in 3.1). This means that the predictor-corrector has its own state.

Definition 2.9 tells us that a predictor-corrector is made up of an *explicit-implicit* pair. It was decided that when constructing a `PredictorCorrectorSolver` object we pass it explicit and implicit solver objects which share the same `ivp` attribute. One issue which arises from this design is that of *ownership*. Since we are interested in the predictor-corrector *pair*, we want to ensure that we are updating the `PredictorCorrectorSolver`'s state at each time step accordingly. We acheive this by giving all solvers a static method `single_iteration` which emulates the behaviour of the existing `compute_next_values` method. The fundamental difference between these methods is that in `single_iteration` we represent our current state by passing method arguments and returning values– rather than updating instance variables (like in `compute_next_values`). Figure 3.2 briefly outlines these interactions.

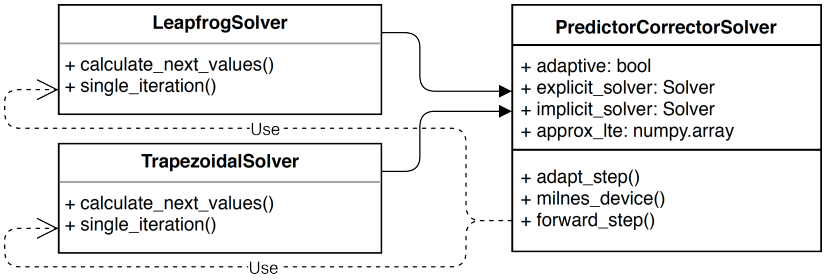

Figure 3.2: UML diagram showing how `PredictorCorrectorSolver` interacts with its explicit (e.g. Leapfrog) and implicit (e.g. Trapezoidal) objects

These method calls all happen inside the `forward_step` method in a PECE manner (2.6). The outline of the `forward_step` method is shown in Listing 3.

```
def forward_step(self, step):
    self.current_time = self.time_mesh[step] \
                      = self.time_mesh[step - 1] + self.next_step_size
    # P: Prediction
    prediction = self.explicit_solver.single_iteration(self.value_mesh,
                                                       self.time_mesh,
                                                       step,
                                                       self.derivative_mesh)
    # E: Evaluation
    derivative = self.ivp.ode.function(prediction, self.current_time)
    self.update_current_state(step_counter, prediction, derivative)
    # C: Correction
    correction = self.implicit_solver.single_iteration(self.value_mesh,
                                                       self.time_mesh,
                                                       step,
                                                       self.derivative_mesh)
    # E: Evaluation
    derivative = self.ivp.ode.function(correction, self.current_time)
    self.update_current_state(step_counter, correction, derivative, lte_approx)
```

Listing 3: Outline of `PredictorCorrectorSolver.forward_step`

3.2.2 Adaptive Step-Size Implementation

To allow for adaptive step-sizes, *Milne's Device* was used to estimate our LTE from a given prediction-correction pair and was implemented in `milnes_device`. We can append Listing 3 with

```
lte_approx = self.milnes_device(prediction, correction)
if self.adaptive:
    self.step_size = self.adapt_step(self.step_size, lte_approx, self.step_tol)
```

Listing 4: Adaptive Step snippet from `PredictorCorrectorSolver.forward_step`

to enable adaptive step computations. The `adapt_step` method uses (2.23) to adjust the step size. This method also accomodates for catastrophic floating point cancellation and ensures that step sizes are not increased by more than double their previous value.

3.3 Graphs & Case Study Animations

It was important to ensure that results from computations could be graphed and reported on in the same way. This would enable coherent and reliable comparisons between methods' behaviours.

For both the n-body problem and the double pendulum, animations were produced to help analyse the results of our computations. In both cases, existing solutions ([11] & [12]) where adapted to better suit our case studies.

The n-body animation uses a modification of the built-in `comet` function in MATLAB. The pendulum and double pendulum animations are built directly from an online example of the python graphing package `matplotlib`.

Chapter 4

Comparison of Methods

Chapter 5

Case Studies

5.1 n-Body Problem

5.2 Pendula

Chapter 6

Reflection

6.1 Deriving the Adaptive 2-step Adams-Bashforth Method

$$e = mc^2 \tag{6.1}$$

Bibliography

[1] U.M. Ascher, L.R. Petzold, "Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations", Philadelphia: SIAM (1935).

[2] I. Newton, "Philosophiae Naturalis Principia Mathematica ("Mathematical Principles of Natural Philosophy")", London (1687).

[3] E. Hairer, S.P. Nørsett, G. Wanner, "Solving Ordinary Differential Equations I", Springer-Verlag Berlin Heidelberg (1993).

[4] D.F. Griffiths, D.J. Higham, "Numerical Methods for Ordinary Differential Equations", Springer-Verlag London Limited (2010).

[5] "https://www.math.utah.edu/software/minpack/minpack/hybrd1.html", "Documentation for MINPACK subroutine HYBRD1", April 2019.

[6] E. Suli, D. Mayers, "An Introduction to Numerical Analysis", Cambridge University Press (2003).

[7] R.D. Gregory, "Classical Mechanics", Cambridge University Press (2006).

[8] W.T. Vetterling, W.H. Press, "Numerical Recipes in C: the art of scientific computing (2nd Edition)", Cambridge University Press (1992).

[9] "http://www.cs.ox.ac.uk/people/pras.pathmanathan/nmood_printable.pdf", "Numerical methods and object-oriented design", January 2019.

[10] A. Iserles, "A First Course in the Numerical Analysis of Differential Equations", Cambridge University Press (1996).

[11] "https://uk.mathworks.com/matlabcentral/fileexchange/67945-multicometfade", "Multiple MATLAB comet trajectory plots", February 2019.

[12] "https://matplotlib.org/gallery/animation/double_pendulum_sgskip.html", "Double Pendulum matplotlib.animation example", March 2019.