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

Joel Biffin

April 7, 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.

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 this 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 trivially 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 the method impacts the stability of a numerical algorithm.

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 local 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 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, $e_i(t) = O(h^p)$ 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. *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 very 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.

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

2.4 Runge-Kutta Methods

2.5 Linear Multistep Methods

2.5.1 Adams Methods

2.5.2 Nystrom Methods

2.5.3 Backwards Differentiation Formulae

2.6 Predictor-Corrector Methods

2.7 Adaptive Step-Size

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).