University of Groningen

Faculty of Science and Engineering

Numerical integration of ODE's with Automatic differentiation

Author:
Nadav Levi
s3318346

Supervisor:
Marcello Seri

Second supervisor:
Hildeberto Jardon
Kojakhmetov

Abstract

In this bachelor project we will look at solving Initial Value Problems of Ordinary Differential Equations by means of numerical integration. Namely, we will be looking at the Taylor method, which approximates a solution of an ODE by using the Taylor series expansion of the function on various points on a grid. One of the main drawbacks of the Taylor method is the need to compute higher order derivatives, which can be computationally expensive as the number of terms grow exponentially. To this end, we present the method of Automatic Differentiation, which is a recursive procedure of generating high-order derivatives. Automatic differentiation bypasses the inefficiency of Symbolic Differentiation and the shortcomings of Numerical Differentiation with respect to round-off and truncation errors. Lastly, we will use the method to evaluate a number of problems from classical mechanics and compare the results to other numerical integrators such as Runge-Kutta.

July 19, 2021

Contents

1 Introduction

Obtaining numerical solutions of ordinary differential equations has a long history that goes back centuries. The first attempt was proposed by Newton, with the recursive computation of the Taylor coefficients, and then with a generalisation by Euler in 1768. In 1824, Cauchy established rigorous error bounds for the Euler method[1].

Newton's idea was to use the Taylor series expansion, which required to successively differentiate y' to obtain

$$y(x_0 + h) = y(x_0) + y'(x_0)h + \frac{y''(x_0)}{2!}h^2 + \cdots + \frac{y^{(p)}(x_0)}{p!}h^p$$

The formulas for higher derivatives soon became increasingly difficult to compute, so Euler proposed to only use a few terms of the Taylor series with h sufficiently small and to repeat the computations from the point of analytical continuation, at $x_1 = x_0 + h$. Euler's method attempted to find a solution for the problem of $y' = f(y), \quad y(x_0) = y_0$ by an approximation of each successive step given by $y_{n+1} = y_n + hf(y_n)$ where y_n approximates $y(x_n)$ and $y(x)$ is the exact solution. However, this series will inevitably contain a truncation error due to omitting the higher order derivatives. There is no right approach, as every method has its own advantages and disadvantages, but in terms of accuracy, a more appropriate approach would be to use Newton's method with a larger p, as this would result in a smaller truncation error compared to Euler's method.

Previously, the Taylor method was regarded as having no practical use.
This was due to the fact that computing the coefficients was difficult and computationally expensive to obtain as the derivatives tended to grow in complexity rather fast.
This was true in particular for composite functions, or functions with multiple dependent variables where terms grew exponentially. Historically, this was used as the main argument against the Taylor method and other methods such as the Runge-Kutta family of methods, which included Euler's method, were believed to be more promising.

However, computer programs used for the evaluation of Taylor series coefficients in an efficient way have been known for almost 5 decades starting from the computer implementation of Moore in [1] using the method of Automatic Differentiation, where it was shown that the Taylor series method in fact required less operations as opposed to the Runge-Kutta method. In fact, after seeing Moore's implementation, Henrici which was initially skeptical of the Taylor method [2], is noted for the following quote referring to the work of Moore:

> "... it is shown how, at least for rational functions the machine can be used to evaluate the total derivatives of f recursively, thus removing the main objection to the Taylor expansion method."

However, this approach only started to gain traction and reach a wide audience in the last decade or so with [3, 4, 5, 6, 7], due to the fact that more and more scientific disciplines required high precision solutions of ordinary differential equations, and standard methods were not able to reach these precision levels. Furthermore, the Taylor method had been used for integration of ODEs in Dynamical Dystems; For instance, in determination of periodic orbits by [8]. Another important application of the Taylor method is that it can be executed using interval arithmetic, which allows us to validate numerical methods for differential equations[6].

In terms of differentiation, some of the most commonly used methods are *Symbolic differentiation*, and *Numerical differentiation*, for example with the *Finite Differences*

method. However, these methods have some shortcomings. Firstly, Numerical differentiation refers to the approximation of derivatives by divided difference. This indicates the presence of truncation errors, which leads to a halving of the significant digits under the best of circumstances[9]. Note that the main motivators for using the Taylor method were to avoid truncation errors due to omitting the higher order derivatives, and due to this fact Numerical differentiation would not be ideal for this purpose. Additionally, Symbolic differentiation(Sometimes referred to as *Computer Algebra*) refers to an automated program, finding the derivative of a given formula with respect to a specified variable. Thus, the program produces a new formula as its output by manipulation of formulas to produce new formulas, rather than performing numeric calculations based on formulas. This results in what is known as *equation swell*, and involves far too many terms for its efficient evaluation[9].

In contrast, the method of *Automatic differentiation* bypasses the truncation errors due to numerical differentiation and the inefficiencies of symbolic differentiation. Automatic differentiation relies on the fact that every computer program can be executed by a series of elementary arithmetic operations, in a way that derivatives of arbitrarily high order can be computed automatically in an accurate way. Thus, the main point of automatic differentiation is based on the idea that a compiler can translate mathematical formulae by applying a set of fixed rules that apply to all functions.

In the following text we will demonstrate how the Taylor method can be used together with Automatic differentiation to obtain numerical solutions for the integration of ordinary differential equations. In section 2, we will begin by describing the theory underlying ordinary differential equations and the Taylor method with a number of definitions and results which will be used throughout this text. Then, in section 2.1 we will describe the Taylor method itself, where we will treat the issue of convergence, and choosing the order and step size of the integrator. In section 2.2, we will introduce the concept of *Hamiltonian systems*, along with *Symplectic integrators*, a class of geometric integrators, which are energy preserving methods, and thus ideal for problems in mechanics. In section 3 the method of Automatic differentiation will be introduced, along with formula translation and the rules for differentiation. Then, we will continue with a problem description in section 4, where three classical problems from Newtonian mechanics will be described; a mathematical pendulum, a double pendulum, and the 2-body problem, after which we will describe the methodology used. In section 5, the results of the simulation using a number of different integrators will be shown. These integrators are: The Taylor method, a symplectic integrator - Leapfrog, which is a $2nd$ order symplectic method; a Runge-Kutta of order 4 method, and the LSODA adaptive integrator from the Python Scipy library. We will then test all four models on the problems described in section 4, and show how the Taylor method performs well against the other models, by preserving the energy of the systems.

2 Preliminaries

Before we begin to address the topic of numerical integration of an ordinary differential equation using the *Taylor method*, a number of basic definitions and results will be presented. These will serve as the foundation of the Taylor and Automatic Differentiation methods and be used throughout this text.
To begin with, we define an ordinary differential equation:

Definition 2.1. An ordinary differential equation of first order is an equation of the form

$$y'(x) = f(x, y(x)) \tag{1}$$

Where $f : [a, b] \to \mathbb{R}^m$ is a smooth function and x as an independent variable such that y' indicates the derivative of y with respect to x. Note that sometimes the x in $y(x)$ is omitted for convenience.
We also consider a set of initial conditions given by $x = x_0$ and $y_0 = y(x_0)$.
A function $y(x)$ is considered a solution of this equation if, for every x,

$$y'(x) = f(x, y(x))$$

However, this terminology is misleadingly simple; note that y can be a vector valued function. Therefore, when working on $\mathbb{R}^n$, and $x \in \mathbb{R}$, the domain and range of f and y are given by:

$$f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n \quad \text{and} \quad y : \mathbb{R} \to \mathbb{R}^n$$

Definition 2.2. An ordinary differential equation of order n is given by

$$f(x, y, y', ..., y^{(n-1)}) = y^{(n)} \tag{2}$$

where f is a function composed of x, y and derivatives of y.

A linear nth order homogeneous differential equation is given by:

$$\mathcal{L}(y) := a_n(x)y^{(n)} + a_{n-1}(x)y^{(n-1)} + \cdots + a_0(x)y = 0. \tag{3}$$

Where a_i is a function of x.

Equation (3) can also be written in the following sense, after dividing by $a_n(x)$:

$$y^{(n)} + b_{n-1}(x)y^{(n-1)} + \cdots + b_0(x)y = 0 \quad \text{where } b_i(x) = \frac{a_i(x)}{a_n(x)} \tag{4}$$

Note that $a_n(x) \neq 0$ away from singular points.

Proposition 2.3. *We consider a differential equation of order n in a similar manner to equation* (2)*. Then, we can transform it into a system of first order equations:*
Set $y_1(x) := y(x), \quad y_2(x) := y'(x)$ *and so on up to* $y_n(x) := y^{(n-1)}(x)$*, such that we obtain n first order equations:*

$$\begin{aligned} y_1'(x) &= y'(x) \\ y_2'(x) &= y''(x) \\ &\vdots \\ y_n' &= y^{(n)}(x) = f(x, y_1, ..., y_n) \end{aligned} \tag{5}$$

Furthermore, if (2) *has n initial conditions given by*

$$y(x_0) = \alpha_1, \quad y'(x_0) = \alpha_2, \quad ..., \quad y^{(n-1)}(x_0) = \alpha_n$$

the system in (5) *has the following initial conditions:*

$$y_1(x_0) = \alpha_1, \quad y_2(x_0) = \alpha_2 \quad ..., \quad y_n(x_0) = \alpha_n$$

Example 2.4. *A mathematical pendulum is a* 2*nd order ODE with the following equation of motion:*

$$m\theta'' + \frac{mg}{l}\sin\theta = 0$$

The pendulum consists of a mass m *attached to an arm of length* l*, and* g *is a gravitational constant. In the pendulum,* θ *is the angle of motion. We can then write it as system of first order differential equations by letting* $\theta' = \omega$*, and dividing by* m *so that we obtain:*

$$\begin{cases} \theta' = \omega \\ \omega' = -\frac{g}{l}\sin\theta \end{cases}$$

Proposition 2.5. *In an more general sense, we consider a linear system of first order ODE's in normal form given by:*

$$\begin{aligned} y_1'(x) &= a_{11}(x)y_1(x) + \cdots + a_{1n}(x)y_n(x) \\ y_2'(x) &= a_{21}(x)y_1(x) + \cdots + a_{2n}(x)y_n(x) \\ &\vdots \\ y_n'(x) &= a_{n1}(x)y_1(x) + \cdots + a_{nn}(x)y_n(x) \end{aligned} \tag{6}$$

Which can be written in matrix form:

$$\begin{pmatrix} y_1' \\ y_2' \\ \vdots \\ y_n' \end{pmatrix} = \begin{pmatrix} a_{11}(x) & a_{12}(x) & \dots & a_{1n}(x) \\ a_{21}(x) & a_{22}(x) & \ddots & a_{2n}(x) \\ \vdots & \vdots & \dots & \vdots \\ a_{n1}(x) & a_{n2}(x) & \dots & a_{nn}(x) \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \tag{7}$$

And then writing in vector form:

$$y'(x) = A(x)y(x) \tag{8}$$

where A *is an* $n \times n$ *matrix and is called the* companion matrix.

If we take the special case of (7) *where the terms in* A *are constants such that* $a_{ij}(x) = a_{ij}$ *and* $A(x) = A$*, the equation becomes fairly simple to solve. We denote by* λ *and* v *the eigenvalues and eigenvectors of* A *respectively. Then, for* $y'(x) = Ay(x)$ *there exists a solution given by*

$$y(x) = exp(\lambda x)v. \tag{9}$$

Yet, this equation can already have complex behaviours, depending on the eigenvalues of the matrix A*. For example, the eigenvalues of* A *can determine the stability behavior of the system around a fixed point. The particular stability behavior depends upon the existence of real and imaginary components of the eigenvalues, along with the signs of the real components and the distinctness of their values[10].*

In contrast, non-linear ordinary differential equations are often hard to solve, and can rarely be solved analytically[11]. This brings the need for numerical methods in order to obtain a solution.

In the following section, as we will describe one type of such numerical method: The *Taylor method*, where we will see how a solution to an ODE given by a function f can be

computed using a sum containing its derivatives. Therefore, an important aspect before describing the method is to establish under which conditions a function is differentiable and whether a solution to an ODE exists.

Definition 2.6. (Real Analytic Function)
Let $E \subset \mathbb{R}$ and define a function $f : E \to \mathbb{R}$.
Then, the function f is *analytic* at point x_0 if there exists an open interval $(x_0-\epsilon, x_0+\epsilon)$ in E for some $\epsilon > 0$ such that there is a power series $\sum_{n=0}^{\infty} c_n(x-x_0)^n$ centred at point x_0 with a radius of convergence $R \geq \epsilon$ converging to f on $(x_0-\epsilon, x_0+\epsilon)$

Theorem 2.7. *Let $E \subset \mathbb{R}$ be an open subset and $f : E \to \mathbb{R}$ a real analytic function. Then, f is infinitely differentiable on E .*

Now that we defined an analytic function and the differentiability condition, we can introduce Taylor's theorem:

Theorem 2.8. ***(Taylor's theorem)*** *Suppose $E \subset \mathbb{R}$ is an open interval and that $f : E \to \mathbb{R}$ is a function of class C^k, i.e. continuous and $k+1$ times differentiable. Then, let $x_0 \in E$ be an interior point of E, and $h \in \mathbb{R}$ such that $x_0 + h \in E$.*
Then, for a point $x_0 \in E$, the $k-$th order Taylor polynomial of f at x_0+h, is the unique polynomial of order at most k, denoted by

$$P_k(x_0+h) = f(x_0) + f'(x_0)h + f''(x_0)\frac{h^2}{2!} + \cdots + f^{(k)}\frac{h^k}{k!} + R(h^k)$$

Where

$$R(h^k) = \mathcal{O}(h^k)$$

Corollary 2.8.1. *The remainder of the Taylor series expansion is bounded such that*

$$\lim_{h\to 0} R(h^k) = 0$$

Remark. For brevity, we will provide the proof for $k = 1$ below. For $k > 1$, a continuation of this proof can be found in [12].

Proof. **Case:** $k = 1$ The Taylor series expansion of y is given by:

$$y(x_0+h) = y(x_0) + y'(x_0)h + R(h)$$

Hence,

$$R(h) = y(x_0) + y'(x_0)h - y(x_0+h)$$

We note that the derivative of $y'(x_0)$ is given by

$$y'(x_0) = \lim_{h\to 0} \frac{y(x_0+h) - y(x_0)}{h}$$

Which shows that the limit of $R(h)$ tends to 0 as $h \to 0$. □

Remark. Often, the remainder of the Taylor series expansion is given in the Lagrange form, and we will also use this form in the following section. Hence, we introduce a simple definition.

Definition 2.9. (Lagrange form of the remainder) Let f be a function such that on the interval containing x_0 and x_0+h, the derivative to the nth order given by $f^{(n)}(x)$ is continuous.
Then, the remainder is given by:

$$f(x_0+h) - \Big(\sum_{i=0}^{n-1} \frac{f^{(i)}(x_0)}{i!}h^i\Big) = \frac{f^{(n)}(c)}{n!}h^n \tag{10}$$

Where $x_0 < c < x_0 + h$ and that we used $x_0 + h - x_0 = h$.

Proposition 2.10. *Let a subset $E \subset \mathbb{R}$ and an interior point x_0 of E.*
Then, we define real analytic function $f : E \to \mathbb{R}$ such that there exists a power series $f(x) = \sum_{n=0}^{\infty} c_n(x - x_0)^n$ for all $(x_0 - \epsilon, x_0 + \epsilon)$ and some $\epsilon > 0$
Then, the derivatives of $f(x)$ satisfy $f^{(k)}(x_0) = k!c_n$ for some integer $k \geq 0$ and in particular, the Taylor series of f around x_0 is given by

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n$$

With the following theorem, we will address under which conditions a solution to a differential equation exists:

Theorem 2.11. *(Picard–Lindelöf) Let the Initial Value Problem(IVP) be as follows:*

$$y'(x) = f(x, y(x)), \text{ and } y(x_0) = y_0$$

Suppose f is uniformly Lipschitz continuous in y, and continuous in x.
Then, for some $\epsilon > 0$ there exists a unique solution $y(x)$ to the initial value problem on the interval $[x_0 - \epsilon, x_0 + \epsilon]$

2.1 Taylor method

In this section we will lay out the foundation for the Taylor method which is used throughout this paper to obtain a numerical scheme for the solutions of ODEs.
The use of Taylor series for the solution of differential equations has been studied by many during the last century. The attraction is due to the inherent power and accuracy of solution methods based on Taylor series. [13]

Taylor methods are based on obtaining the Taylor polynomial of function f, as in Theorem 2.8, by expanding each component of the solution in a long series[14].
This procedure involves differentiation of f to a chosen order p.

We will now consider a system of ordinary differential equations in explicit form as given in (1).

Theorem 2.12. *Let E denote an interval given by $[x_0, x_{max}] \subset \mathbb{R}$ and let f be an analytic function given by $f : E \times \mathbb{R}^n \to \mathbb{R}^n$. Additionally, let $y : E \to \mathbb{R}^n$ be a solution to the IVP such that*

$$\begin{cases} y'(x) = f(x, y(x)) \\ y(x_0) = y_0 \end{cases}$$

Then, a solution is given by:

$$y(x_0 + h) = y(x_0) + y'(x_0)h + \frac{y''(x_0)}{2!}h^2 + \dots$$

With the Taylor method we then approximate the solution of $y'(x)$ for $x_0 + h \in E$ as the truncated Taylor expansion of the solution around the previous point, x_0, given by:

$$y(x_0 + h) = y(x_0) + y'(x_0)h + \frac{y''(x_0)}{2!}h^2 + \cdots + \frac{y^{(p)}(x_0)}{p!}h^p + \mathcal{O}(h^{p+1}) \tag{11}$$

Where $h = x_0 + h - x_0$ is the integration time step, and $y^{(i)}(x)$ denotes the $i-$th derivative of $y(x)$ while p denotes the order of the Taylor method.

We then define the n-th normalised derivative as:

$$y^{[n]}(x) = \frac{1}{n!}y^{(n)}(x)$$

such that equation (11) simplifies to:

$$\begin{aligned} y(x_0 + h) &= y(x_0) + y'(x_0)h + \frac{y''(x_0)}{2!}h^2 + \cdots + \frac{y^{(p)}(x_0)}{p!}h^p + \mathcal{O}(h^{p+1}) \\ &= \sum_{i=0}^{p} h^p y^{[p]}(x_0) + \mathcal{O}(h^{p+1}) \end{aligned} \tag{12}$$

The idea of the method is relatively simple: Given an initial condition $y(x_n) = y_n$, the next value $y(x_{n+1})$ is approximated from the Taylor series of $y(x)$ at the previous point.

One of the main considerations for practical implementation is the computation of the values of the n-th order derivatives of $y^{(p)}(x_0)$. A simple procedure to obtain these is to differentiate equation (1) with respect to the independent variable x so that we obtain

$$\begin{aligned} y'(x) &= f(x, y(x)) \\ y''(x) &= f_x + f_y y'(x) = f_x + f_y f(x) \\ y'''(x) &= f_{xx} + 2f_{xy}f(x) + f_{yy}f(x)^2 + f_y(f_x + f_y f(x)) \\ &\vdots \\ y^{(p)}(x) &= F(x, y, y', ..., y^{(p-1)}) \end{aligned} \tag{13}$$

Example 2.13. *As a concrete example, take the following function $y(x) = x^4 + x$ evaluated at $x_0 = 3$*
Then, this can be evaluated by the Taylor series in the following manner:

$$\begin{aligned} y(x_0 + h) &= y(x_0) + y'(x_0) + \frac{y''(x_0)}{2!}h^2 + \frac{y'''(x_0)}{3!}h^3 + \frac{y''''(x_0)}{4!}h^4 + \frac{y'''''(x_0)}{5!}h^5 + \dots \\ &= (x_0^4 + x_0) + (3x_0^3 + 1) + (\frac{12x_0^2}{2!}h^2) + (\frac{24x_0}{3!}h^3) + (\frac{24}{4!}h^4) + 0 \\ &= 84 + 109h + \frac{108}{2!}h^2 + \frac{72}{3!}h^3 + \frac{24}{4!}h^4 \\ &= 84 + 109h + 54h^2 + 12h^3 + h^4 \end{aligned} \tag{14}$$

Now, in order to verify with result, we can simply plug $x_0 + h = 3 + h$ into $y(x) = x^4 + x$ to obtain the following

$$\begin{aligned} y(3 + h) =& (3 + h)^4 + (3 + h) \\ &= (9 + 6h + h^2)^2 + 3 + h \\ &= 84 + 109h + 54h^2 + 12h^3 + h^4 \end{aligned}$$

Which agrees with the result found by the Taylor series expansion. The main tasks is therefore to obtain the higher order derivatives of the function. In the basic example above, finding the derivative is quite straightforward, as the derivatives vanish after $p = 4$, and in any case, the function is a simple polynomial which we can differentiate by applying the basic rules of calculus. However, evaluating high order derivatives is not always as straightforward to compute for compound functions and whenever the derivatives contain partial differentiation as in (13)*, where the terms grow exponentially.*

2.1.1 Convergence

In the following subsection a number of definitions and propositions will be presented which will be useful in establishing the conditions necessary for a Taylor series to converge.

Definition 2.14. Inclusion Isotonicity:
Let the sets $Y, X \subset \mathbb{R}^n$ be subsets of dimension n and a function $F : \mathbb{R}^n \to \mathbb{R}^n$.

An interval is said to be *inclusion isotonic interval* if for each $i \in \mathbb{N}$, the following relation applies:

$$Y_i \subseteq X_i \Longrightarrow F(Y_1, \ldots . Y_n) \subseteq F(X_1, \ldots, X_n)$$

Definition 2.15. Let X_0 be an interval and y a real-valued function of the real variable x, with $y(x)$ defined for all $x \in X_0$. An *interval enclosure* of y is an inclusion isotonic interval-valued function Y of an interval variable X , with $Y(X)$ defined for all $X \subseteq X_0$, having the property that

$$y(x) \in Y(x) \quad \text{for all } x \in X_0$$

Remark. A non-degenerate interval is an interval that contains more than one point.

Proposition 2.16. *Suppose $y(x)$ is analytic in some neighborhood of x_0*
For the finite Taylor expansion with Lagrange form of the remainder, we have

$$y(x) = \sum_{i=0}^{n-1} y^{[i]}(x - x_0)^i + y^{[n]}(s)(x - x_0)^n \quad \textit{for some } s \in [x_0, x] \tag{15}$$

If R_n in an interval enclosure of $y^{[n]}(s)$ such that

$$y^{[n]}(s) \in R_n([x_0, x])$$

we have

$$y(x) \in \sum_{i=0}^{n-1} y^{[i]}(x - x_0)^i + R_n([x_0, x])(x - x_0)^n$$

where $R_n([x_0, x])(x - x_0)^n$ is an enclosure for the Taylor series.

Example 2.17. *Let $y(x)$ be an analytic function for $x \in X_0$.*
Then, the interval polynomial below is an enclosure of x.

$$Y_n(x) = \sum_{i=0}^{n-1} y^{[i]}(x - x_0)^i + R_n([x_0, x])(x - x_0)^n$$

Where we have $y(x) \in Y_n(x) \in Y_n(X)$ for non-degenerate intervals $X \subset X_0$

Example 2.18. *As a more specific example, we take $y(x) = e^x$, $x_0 = 0$, $X_0 = [0, 1]$ and $n = 2$. Then, we obtain*

$$e^x \in 1 + x + \frac{1}{2}[1, e]x^2 = [1 + x + \frac{1}{2}x^2, 1 + x + \frac{e}{2}x^2] \quad \textit{for all } x \in [0, 1]$$

Proposition 2.19. *The Taylor series given in Theorem 2.12 converges for all*

$$|h| < r(1 - \exp(-1/2M)) \quad \textit{where } |x|, |y| \leq r \quad \textit{and } M = \max_{|x|,|y|\leq r} |f(x, y)|$$

Proof. This proof outlines the main ideas of Cauchy's convergence proof for the Taylor series expansion of a function and follows the steps given in [15].

Suppose that the function $f(x, y)$ is analytic around the initial values given by (x_0, y_0). For simplicity, we assume that the initial values are located at the origin, so that $f(x, y)$ can be written in the following sense:

$$f(x, y) = \sum_{i,j\geq 0} a_{ij} x^i y^j \tag{16}$$

where a_{ij} are the coefficients of the partial derivatives occurring in $y', y'', y''', ...$
Additionally, we assume that the series above converges for $|x|, |y| < r$.
This implies that

$$|a_{ij}| \leq \frac{M}{r^{i+j}}$$

The rest of the proof uses the *method of majorants*, and relies on the fact that all the coefficients in $y^{(i)}$ are positive, so that we can replace a_{ij} by the largest possible value given by M, to obtain the worst possible result:

$$f(x, y) \leq \sum_{i,j\geq 0} M \frac{x^i y^j}{r^{i+j}} = \frac{M}{(1 - \frac{x}{r})(1 - \frac{y}{r})}$$

Following that, $f(x, y) = y'$ such that the majorizing differential equation becomes:

$$y' = \frac{M}{(1 - \frac{x}{r})(1 - \frac{y}{r})}, \quad y(0) = 0$$

Which can be easily integrated, for instance by separation of variables, which yields:

$$y = r\Big(1 - \sqrt{1 + 2M \log(1 - \frac{x}{r})}\Big)$$

This solution, has a power series expansion that converges for every x. Thus,

$$|\sqrt{2M \log(1 - x/r)}| < 1$$

which proves our proposition. □

2.1.2 Choosing the order size p and integration time step h

In contrast to other methods which use a fixed-order, a Taylor integrator allows the user to choose both the order size given by p, as well as the integration time-step given by h. This is due to the fact that different combinations of p and h can give the same final integration error, so there is a considerable flexibility in the way p and h can be chosen. In this section we are mainly going to follow the ideas presented by [4] and later on by Biscani et al [5] .

Example 2.20. *Evidently, different values of p and h can lead to very different propagation of round-off errors, even with the same truncation error, while evaluating the Taylor polynomial.*

To illustrate this point, consider the following differential equation, which was suggested by [4]:

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

Furthermore, we let the time interval be between 0 *and* 10π.

We know that a solution of $y(x)$ *is given by* $y(x) = \cos(x)$.
Thus, if we are interested in calculating $y(10\pi)$ *by numerically integrating the differential equation above using the Taylor series expansion, with an error below* 10^{-15}, *we can begin by evaluating the expansion at* $x_0 + h$ *to obtain the following:*

$$\begin{aligned} y(x_0 + h) &= y(0) + y'(0)h + \frac{y''(0)}{2!}(h)^2 + \cdots + \frac{y^{(p)}(0)}{p!}(h)^p \\ &= 1 - \frac{h^2}{2!} + \frac{h^4}{4!} - \frac{h^6}{6!} + \frac{h^8}{!8} - \dots \\ &= \sum_{i=0}^{n} \frac{(-1)^i}{2} \frac{h^{2i}}{i!} \end{aligned}$$

Where we used the fact the every odd term of the sine function is equal to 0 *at the point of evaluation of an even multiple of* π. *Due to the properties of this function, a step size* $h = 10\pi$ *can be used which would require a single step on the time interval, along with a sufficiently high order* p. *Jorba and Zou argue that it is relatively straightforward to verify that up to* $p = 95$ *the series above would yield a truncation error that is lower than the required* 10^{-15} *[4]. However, they find that by applying Newton–Raphson method with double precision arithmetic, this series results in a large error.*
The problem here, is that the series contains odd terms that vanish; i.e., there are large terms that cancel out which subsequently results in the loss of many significant digits when the sum is carried out with floating point arithmetic.

However, this issue can be overcome by using a smaller step size, so that such cancellations are avoided. This example comes to demonstrate the importance of choosing the right step size h.

As we have shown above, choosing the step size h can lead to very different errors when evaluating a function with the Taylor method. This raises the question on how to select the optimal step size h and the order p of the Taylor series. As we have shown in proposition (2.19), the Taylor series converges for $|h| < r(1 - \exp(-1/2M))$. So any choice of h will have to satisfy this condition. There are various approaches on choosing the step size h. For example, in [14], they showed how the truncation error of a Taylor series can be controlled by giving upper and lower bounds for the error for functions with only one singularity on the circle of convergence, and for functions with two or more primary singularities by taking a small step h in the complex plane directly toward one of the primary singularities and extending the series by analytic continuation. Furthermore, there are strategies to use step sizes that are larger than the radius of convergence of the series, but they only work for some singularities and require some computational effort[13]. However, in this section, we will follow the approach given by Jorba and Zou in [4] for controlling the error by choosing the optimal h and p.

We begin by observing that the power series expansion of the solution $y(x_m)$ can have different radii of convergence for different values of x_m, which needs to be taken into account. Essentially, this means that at every time step we need to compute appropriate values for the order size and time step given by $p = p_m$ and $h = h_m$ for every m.
Assuming that for a given point x_m, the solution is at point y_m such that $y(x_m) = y_m$

and we would like to compute the position at point $x_{m+1} = x_m + h_m$ for a certain error, denoted by ϵ.
Then, assuming $h_m = x_{m+1} - x_m$ is small, we have the following:

$$y_m(x) = \sum_{i=0}^{\infty} y_m^{[i]}(x_m)h^i$$

And we would like to choose a relatively small value that would minimise h_m and a large enough value for p, so that we then obtain

$$\|y_m(x_{m+1}) - y_{m+1}\| \leq \epsilon$$

Where

$$x_{m+1} = x_m + h_m, \quad y_{m+1} = \sum_{i=0}^{p} y_m^{[i]}(x_m)h_m^i$$

Furthermore, the total number of operations used for the numerical integration has to be as small as possible. In order to find such values, we need to make further assumptions on the solution of y(x), and in particular assumptions about the analyticity.
For that, we will outline the proposition suggested in [4], building on previous work of Simó et al [16], where they demonstrate that in the context of Taylor methods for integration, there exists an optimal choice for p, if our objective is to improve the performance of the integrator, which would be done by minimising the amount of floating-point operations per unit of integration time.

Proposition 2.21. *Assume that the function $h \mapsto y(x_m + h)$ is analytic on a disk of radius R_m, and that there exists a positive constant K_m such that for every $i \in \mathbb{N}$ we have:*

$$|y_m^{[i]}| \approx \frac{K_m}{R_m^i}$$

Furthermore, as long as the required accuracy $\epsilon \longrightarrow 0$, the optimal value of h that minimised the number of operations tends to

$$h_m = \frac{R_m}{e^2}$$

and the optimal order, denoted by p_m, behaves like

$$p_m = -\frac{1}{2}\ln\left(\frac{\epsilon}{K_m}\right) - 1 \tag{17}$$

For further properties and proof of this proposition, see [4] and [16]

Remark. Note that the optimal step size does not depend on the level of required accuracy ϵ. In fact, the optimal order is the order that guarantees the required precision once the step size has been selected.

One of the biggest drawbacks of Proposition 2.21 is that it requires us to know in advance the radius of convergence of the Taylor series or the value of the constant K_m which we cannot obtain easily.
To get around that, we will outline the method described by Jorba and Zou [4] where they allow the user to choose the desired integration accuracy by introducing a term for the the absolute tolerance and relative tolerance.
Let the error term be denoted by either the absolute tolerance for the error ϵ_a or the relative tolerance for the error ϵ_r, namely:

$$\epsilon_m = \begin{cases} \epsilon_a & \text{if } \epsilon_r\|y_m\|_\infty \leq \epsilon_a \\ \epsilon_r & \text{otherwise} \end{cases}$$

Remark. Note that the errors above are chosen by the user of the program.

Subsequently, the order size becomes:

$$p_m = \left\lceil -\frac{1}{2}\ln(\epsilon) + 1 \right\rceil$$

Where $\lceil \cdot \rceil$ is the ceiling function. Comparing this with Proposition 2.21 we note that the positive constant was chosen to be $K_m = 1$ and that p_m in equation (17) is two units larger.

In a similar manner, the radius of convergence R_m can also be distinguished into two cases depending on the chosen error:

$$R_m^i = \begin{cases} \left(\frac{1}{\|y_m^{[i]}\|_\infty}\right)^{\frac{1}{i}} & \text{if } \epsilon_r \|y_m\|_\infty \leq \epsilon_a \\ \\ \left(\frac{\|y_m\|}{\|y_m^{[i]}\|_\infty}\right)^{\frac{1}{i}} & \text{otherwise} \end{cases}$$

Where in any case, the radius of convergence is estimated to be the minimum of the last two terms

$$R_m = min\{R_m^{(p-1)}, R_m^p\}$$

And lastly, the estimated time step becomes

$$h_m = \frac{R_m}{e^2}$$

In their implementation, Jorba and Zou [4] introduce a safety factor depending on the order p for $h_m = \frac{R_m}{e^2}$ which is given by

$$h_m = \frac{R_m}{e^2}\exp\left(-\frac{0.7}{p_m - 1}\right)$$

due to the fact that the calculations used to obtain h_m were based on asymptotic estimates.

However, in the work of Biscani and Izzo [5] with the Heyoka package where they continue the work of Jorba and Zou, this distinction is not followed. In contrast, they impose $\epsilon_r = \epsilon_a$ so that the user only needs to specify one value for the error tolerance. They then continue to describe how in relative error control mode, where $\|y_0\|_\infty > 1$, leads to the radius of convergence given by

$$R_m = \left(\frac{\|y_0\|_\infty}{\|y^{[i]}(x_0)\|_\infty}\right)^{\frac{1}{i}}$$

Hence, R_m becomes an estimation of the smallest convergence radius re-scaled by the largest value in the state vector.
Furthermore, It can be shown that, if p, h and R_m are chosen in the way outline above, the error resulting from the truncation of the Taylor series expansion will be bounded in either an absolute or relative sense, by the user-defined tolerance. In our implementation, we will using the Heyoka package for the Taylor method, and thus will be following the implementation of Biscani and Izzo.

2.2 Symplectic integrators

Hamiltonian systems are an important class of ordinary differential equations describing energy preserving physical systems.
We will briefly describe the theory of Hamiltonian systems following the work done by Hairer [17].

Definition 2.22. Let $H(x,p)$ be a scalar sufficiently differnetiable function. Then, for vectors x and p in $\mathbb{R}^d$ we have

$$\dot{x} = -\nabla_p H(p,x), \quad \dot{p} = \nabla_x H(x,p) \tag{18}$$

Where H is called the 'Hamiltonian' and ∇ denotes the partial derivative of H with respect to one of the dependent variables.

Furthermore, if the Hamiltonian system can be described as

$$H(x,p) = \frac{1}{2}p^2 + V(x)$$

then Hamilton's equations encode Newton's equation:

$$\begin{cases} \dot{p} = -\nabla_x V \\ \dot{x} = p \end{cases}$$

implying

$$\ddot{x} = -\nabla_x V$$

Remark. The Hamiltonian H is an integral of motion that describes the total energy of a system.

Definition 2.23. A non constant function $I(y)$ is a *first integral* of $\dot{y} = f(y)$ if

$$I'(y)f(y) = 0 \quad \forall y.$$

Or alternatively, this definition says that every solution $y(t)$ of $\dot{y} = f(y)$ satisfies the following condition:

$$I(y(t)) = c \quad \text{Where c is constant}$$

Proposition 2.24. *The Hamiltonian system given by $H(p,x)$ in (18) , is a first integral, and as such, conserves the total energy.*

We will now begin to treat the concept of *Symplectic integrators*, that form a subclass of *Geometric integrators*. A numerical method is called geometric if it preserves one or more physical or geometric properties of a system, up to round-off errors. Therefore, geometric integrators preserve geometric, structural and physical properties of the considered differential equations. For example, some properties that can be preserved are the symplectic structure, symmetries, etc. Therefore, geometric methods have applications in many areas of physics, including celestial mechanics, particle accelerators, molecular dynamics, fluid dynamics, pattern formation, plasma physics, reaction- diffusion equations, and meteorology[18].

Symplectic integrators are designed for the numerical solution of Hamilton's equations given in (18). For a more extensive overview of the subject, refer to Hairer [17].

Remark. An important property of Symplectic Integrators is that the energy of the system is preserved.

Besides symplectic integrators, another class of integrators are the so called *Discrete gradient numerical schemes*, which were introduced in order to numerically integrate N-body systems of classical mechanics with possible applications to molecular dynamics and celestial mechanics[19]. In the following section, a brief introduction will be given.

2.2.1 Non-standard discrete gradient schemes

Consider the one-dimensional Hamiltonian system given by (18). Then, a non-standard discrete gradient scheme, can be described in the following way:

$$\begin{aligned} \frac{x_{n+1}-x_n}{\delta_n} &= \frac{H(x_{n+1},p_{n+1})+H(x_n,p_{n+1})-H(x_{n+1},p_n)-H(x_n,p_n)}{2(p_{n+1}-p_n)} \\ \frac{p_{n+1}-p_n}{\delta_n} &= \frac{H(x_n,p_{n+1})+H(x_n,p_n)-H(x_{n+1},p_{n+1})-H(x_{n+1},p_n)}{2(x_{n+1}-x_n)} \end{aligned} \tag{19}$$

where δ_n is an arbitrary positive function of $h, x_n, p_n, x_{n+1}, p_{n+1}$
In the separable case where $H = T(p) + V(x)$ the scheme reduces to:

$$\begin{aligned} \frac{x_{n+1}-x_n}{\delta_n} &= \frac{T(p_{n+1})-T(p_n)}{p_{n+1}-p_n} \\ \frac{p_{n+1}-p_n}{\delta_n} &= \frac{V(x_{n+1})-V(x_n)}{x_{n+1}-x_n} \end{aligned} \tag{20}$$

Furthermore, the system in (19) is a consistent approximation of the Hamiltonian system in (18) if the following condition holds:

$$\lim_{h\to 0}\frac{\delta_n}{h} = 1 \tag{21}$$

Theorem 2.25. *The numerical scheme in (19) preserves the energy integral such that* $H(x_{n+1}, p_{n+1}) = H(x_n, p_n)$

Corollary 2.25.1. *An n-th order Taylor method for numerical integration of a Hamiltonian system is constructed in a similar manner to the scheme given in (19) and as such would preserve the total energy of a system.*

Remark. A proof of Theorem 2.24 and the following corollary can be found in [19]. The main idea is that the system in (19) implies the equality of both numerators on the right-hand side of the equation, which would prove the theorem.

With this property, the Taylor method would also conserve the energy of a system in a similar manner to symplectic integrators as long as the property in (21) is satisfied. In turn, this would make the Taylor method competitive for problems where preservation of energy is required.

3 Automatic Differentiation

Automatic differentiation is a widely used method in mathematics and computer science by which a recursive procedure is used to numerically evaluate a derivative of a certain function by repeatedly applying the chain rule and the rules of elementary calculus. Derivatives are commonly used in science and engineering, which brings the need for an efficient method to do so. There are a number of popular methods for obtaining derivatives, such as *Symbolic differentiation* and *Numerical differentiation*; their success relies on evaluating the derivatives accurately and efficiently. The method of automatic differentiation relies on the fact that every computer program can be executed by a series of elementary arithmetic operations, in a way that derivatives of arbitrarily high order can be computed automatically in an accurate way. Thus, the main point of automatic differentiation is based on the idea that a compiler can translate mathematical formulae by applying a set of fixed rules that apply to all functions. The functions considered are those that can be obtained by elementary operations and by composition of other elementary functions. These include polynomials, exponential and trigonometric functions among others.[4]

3.1 Formula translation

Differentiation of a function defined by a formula depends on the translation of that formula into a list of instructions for a sequence of executable operations.
Therefore, in order for a compiler to evaluate such a function, which for illustration purposes, we consider the following Bernoulli equation:

$$y' = f(x, y) = \frac{2y}{x} - x^2 y^2 \tag{22}$$

It must be written in the form:

$$F = (2 * (Y/X) - (X * *2) * (Y * *2)) \tag{23}$$

We will focus mainly on functions that can be evaluated by performing a sequence of arithmetic operations. Such functions are termed *Codeable functions* and the set of elementary operations along with their symbolism can be any of those which are listed in the table below. These operations and functions can be nested in any combination to any degree of complexity[9].

Operators and functions	
Symbol	Operation
+, -, *, /, **	Addition, subtraction, multiplication, division, power
SQRT	Square root
EXP, LOG	Exponential, logarithm
SIN, COS, TAN	Trigonometric functions

The values of the variables and constants are considered here to be the data, while the value of the function F is the output of the calculation.
An analysis of a formula for a considered function, with a series of arithmetic operations, allows the user to obtain an equivalent representation of the desired function into a code

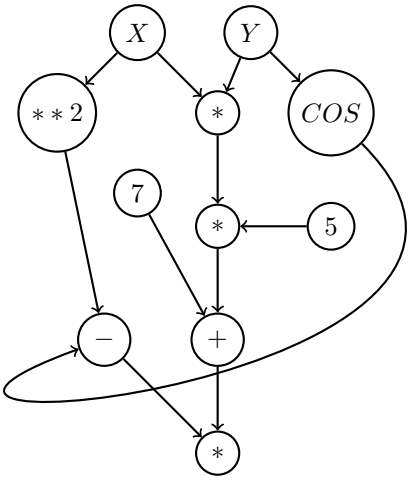
list[20].
To illustrate this process, consider the function given above, which can be represented by the series of instructions shown in the following code list:

$$\begin{cases} u_1 = x \\ u_2 = y \\ u_3 = u_2/u_1 \\ u_4 = 2 * u_3 \\ u_5 = u_1 * u_1 \\ u_6 = u_2 * u_2 \\ u_7 = u_5 * u_6 \\ u_8 = u_4 - u_7 \\ y' = u_8 \end{cases} \tag{24}$$

Note that each line in (24) contains only a binary operation on two input data or previously computed value, and that the code list is in itself a series of statements written in the same language as formula (23), and as a result can be translated into a machine language by a compiler for execution. Note that this list is not unique, and a particular function can be translated into different lists.

3.2 Kantorovich graph of a codeable function

Generally, formula translation presents a slightly greater challenge than differentiation[20]. The importance of the code list is evident from looking at the example given in (24), where each u_i represents a fairly simple function to differentiate. Automatic differentiation relies on reducing a function into its basic building blocks, represented by the code list. Hence, producing the list is a major part in automatic differentiation. However, one challenge is that in formula translation, all rules must be made explicit and must apply to every function. Therefore, such a program must always produce the same series of instructions when applied to the same formula.
Consequently, representing the list as a graph is a useful method for doing so. Such a graph is called the *Kantorovich Graph* of function f and is illustrated below for $f(x, y) = (x^2 - \cos y)(5xy + 7)$



Automatic differentiation is therefore a procedure of traversing a Kantorovich graph, where each node represents an element from the code list, and is paired with its derivative. Hence, for every node u_i, we obtain a tuple $(u_i, \dot{u}_i)$.

3.3 Rules of differentiation

The rules for differentiation, in contrast to formula translation, are known explicitly from calculus. The basic idea underlying differentiation rules is simple; once a function is evaluated and the code list has been obtained we proceed to differentiate each element of the code list, representing nodes on the graph, on line by line basis. By following this procedure, each intermediate element(Primal) is coupled with its partial derivatives(Tangents)

In order to simplify the discussion, we introduce the following notation:

- $u : x \in I \subset \mathbb{R} \mapsto \mathbb{R}$
- As the normalised derivative of $y(x)$ w.r.t. x to the n-th order

$$Y^{[n]} = \frac{1}{n!} y^{(n)}(x_0)$$

- As the Taylor coefficient of $f(x, y(x))$.

$$F^{[n]} = \frac{1}{n!} (f(x, y(x)))^{(n)} \Big|_{x=x_0}$$

- Then, the Taylor series expansion is :

$$Y(x_0 + h) = \sum_{n=0}^{\infty} h^n Y_n$$

- Such that the normalised derivative to order $(n + 1)$ becomes:

$$Y^{[n+1]} = \frac{1}{n+1} F^{[n]}$$

Then, as a generalisation of the rules of differentiation from elementary calculus, we introduce the following proposition for the rules of automatic differentiation for higher order derivatives:

Proposition 3.1. *Rules for Automatic Differentiation*
For $n = 0, 1, 2...$ *:*

1. *If* $u = p \pm q$:

$$u^{[n]} = p^{[n]} \pm q^{[n]} \tag{25}$$

2. *If* $u = p * q$:

$$u^{[n]} = \sum_{i=0}^{n} p^{[i]} q^{[n-i]} \tag{26}$$

3. *If* $u = p/q$:

$$u^{[n]} = \frac{1}{q^{[0]}} \left(p^{[n]} - \sum_{i=1}^{n} u^{[n-i]} q^{[i]} \right) \tag{27}$$

4. *If* $u = \exp(p)$

$$u^{[0]} = \exp\left(p^{[0]}\right) \text{ and } u^{[n]} = \frac{1}{n!} \sum_{i=0}^{n-1} (n-i) u^{[i]} p^{[n-i]} \tag{28}$$

5. *If* $u = \log(p)$

$$u^{[0]} = \log\left(p^{[0]}\right) \text{ and } u^{[n]} = \frac{1}{p^{[0]}} \left(p^{[n]} - \frac{1}{n!} \sum_{i=1}^{n-1} (n-i) p^{[i]} u^{[n-i]} \right), \tag{29}$$

6. *If* $u = p^c$, *where* c *is a constant such that* $c \neq 1$

$$u^{[0]} = p^c \text{ and } u^{[n]} = \frac{1}{n p^{[0]}} \sum_{i=0}^{n-1} \left(cn - (c+1)i \right) u^{[i]} p^{[n-i]} \tag{30}$$

7. $u = cos(p)$ *and,* $s = sin(p)$

$$u^{[0]} = \cos\left(p^{[0]}\right) \text{ and } u^{[n]} = -\frac{1}{n!} \sum_{i=0}^{n-1} (n-i) s^{[i]} p^{[n-i]} \tag{31}$$

$$s^{[0]} = \sin\left(p^{[0]}\right) \text{ and } s^{[n]} = \frac{1}{n!} \sum_{i=0}^{n-1} (n-i) u^{[i]} p^{[n-i]} \tag{32}$$

Proof. We will provide a short proof for proposition 3.1. A more detailed proof can be found in the literature, for instance in [4], [1] and [15].

Using $u(x) = F(p(x), q(x))$

1. Using the Algebraic Differentiability Theorem,

$$\begin{aligned} u'(x) &= \lim_{x \to c} \frac{u(x) - u(c)}{x - c} \\ &= \lim_{x \to c} \frac{p(x) + q(x) - p(c) - q(c)}{x - c} \\ &= \lim_{x \to c} \frac{p(x) - p(c)}{x - c} + \lim_{x \to c} \frac{q(x) - q(c)}{x - c} \\ &= p'(x) + q'(x) \end{aligned}$$

   Remark. Note that by mathematical induction this applies up to any arbitrary n-th order.

2. Applying Leibniz's formula given by $(fg)^{(n)} = \sum_{i=0}^{n} \frac{n!}{(n-i)!i!} f^{(n-i)} g^{(i)}$ we obtain:

$$\begin{aligned} u^{[n]} &= \frac{1}{n!} u^{(n)} \\ &= \frac{1}{n!} (p * q)^{(n)} \\ &= \frac{1}{n!} \sum_{i=0}^{n} \frac{n!}{(n-i)!i!} p^{(n-i)} q^{(i)} \\ &= \sum_{i=0}^{n} p^{[n-i]} q^{[i]} \end{aligned}$$

3. Writing $u(x) = \frac{p(x)}{q(x)}$ as $u(x)q(x) = p(x)$ we can apply the result from (2)

4. Writing $u(x) = e^{p(x)}$, we take the logarithm and differentiate to obtain $u' = p'u$. We then use (2) to obtain the result.

5. Writing $u = log(p)$, we differentiate both sides to obtain $u' = \frac{p'}{p}$ and use (3) to obtain the result.

6. We take logarithms on both sides and differentiate to obtain $u'p = cup'$ and we use (2) and (3) to obtain the result.

7. We take derivatives on both sides for u, s to obtain $u' = -sin(p)p' = -sp'$ and similarly for $s = sin(p)$ we obtain $s' = up'$. We then use (2) to obtain the required result.

□

Remark. In general, we are able to program a procedure for applying the rules outlined above in order to compute the Taylor coefficients of a function $y(x)$ in the following manner:

1. Represent the function $y(x)$ and consecutive derivatives $y'(x), y''(x) \dots y^{(n)}$ by a finite code list that consists of binary or unary operations.

2. We then generate subprograms for the Taylor coefficients on a line-by-line basis for each item in the list, using the appropriate rule from the proposition above.

3. We finally organize the subprograms and the data such that the program will accept initial values for y, x and the n-th order required.
   The program will then evaluate and store all the Taylor coefficients beginning from 1 up to order n.

Example 3.2. *Consider the Bernoulli equation described in (23), given by:*

$$y' = \frac{2y}{x} - x^2 y^2$$

Evaluating this equation into a workable code list yields:

$$\begin{cases} u_1 = x \\ u_2 = y \\ u_3 = u_2/u_1 \\ u_4 = 2 * u_3 \\ u_5 = u_1 * u_1 \\ u_6 = u_2 * u_2 \\ u_7 = u_5 * u_6 \\ u_8 = u_4 - u_7 \\ y' = u_8 \end{cases} \tag{33}$$

Then, we apply the rules of Proposition 3.1 to each element in the code list to obtain the list of derivatives to order n:

$$\begin{cases} u_1^{[n]} = x^{[n]} \\ u_2^{[n]} = y^{[n]} \\ u_3^{[n]} = \frac{1}{u_1^{[0]}}(u_2^{[n]} - \sum_{i=1}^{n} u_3^{[n-i]} u_1^{[i]}) = \frac{1}{x^{(0)}}(y_2^{(n)} - \sum_{i=1}^{n} (\frac{x}{y})^{(n-i)} x^{(i)}) \\ u_{[4]}^{[n]} = 2 * u_3^{[n]} = \frac{2}{x^{(0)}}(y_2^{(n)} - \sum_{i=1}^{n} (\frac{x}{y})^{(n-i)} x^{(i)}) \\ u_{[5]}^{[n]} = \sum_{i=0}^{n} u_1^{[i]} * u_1^{[n-i]} = \sum_{i=0}^{n} x^{(i)} * x^{(n-i)} \\ u_{[6]}^{[n]} = \sum_{i=0}^{n} u_2^{[i]} * u_2^{[n-i]} = \sum_{i=0}^{n} y^{(i)} * y^{(n-i)} \\ u_{[7]}^{[n]} = \sum_{i=0}^{n} u_5^{[i]} u_6^{[n-i]} \\ u_{[8]}^{[n]} = u_4^{[n]} - u_7^{[n]} \\ y^{[n+1]} = \frac{1}{n+1} u_8^{[n]} \end{cases} \tag{34}$$

In order to evaluate the accuracy of our result, we can continue with a verification that involves using the formulae above, and compare the result with the exact derivative of y which can be obtained by manually taking partial derivatives of it.

We begin by finding the first order derivatives of each element in the list:

$$\text{Let } n=1: \begin{cases} u_1^{[1]} = x^{[1]} = 1 \\ u_2^{[1]} = y^{[1]} = u_8 \\ u_3^{[1]} = (\frac{1}{u_1})(u_2^{[1]} - u_3 * u_1^{[1]}) = \frac{y'}{x} - \frac{y}{x^2} \\ u_4^{[1]} = 2 * u_3^{[1]} = \frac{2y'}{x} - \frac{2y}{x^2} \\ u_5^{[1]} = \sum_{i=0}^{1} u_1^{[i]} u_1^{[1-i]} = xx' + x'x = 2x = 2 * u_1 \\ u_6^{[1]} = \sum_{i=0}^{1} u_2^{[i]} u_2^{[1-i]} = yy' + y'y = 2yy' = 2 * u_2 u_2^{[1]} \\ u_7^{[1]} = \sum_{i=0}^{1} u_5^{[i]} u_6^{[1-i]} = u_5 u_6^{[1]} + u_5^{[1]} u_6 = 2x^2 yy' + 2xy^2 \\ u_8^{[1]} = u_4^{[1]} - u_7^{[1]} = \frac{2y'}{x} - \frac{2y}{x^2} - 2x^2 yy' - 2xy^2 \\ y^{[2]} = \frac{u_8^{[1]}}{2!} = \frac{y''}{2} = \frac{y'}{x} - \frac{y}{x^2} - x^2 yy' - xy^2 \end{cases}$$

We can continue with this procedure by computing the values for $n = 2, 3, ...$ by using the previously computed values up to the required order, which will allow us to find the normalised derivatives of y

Remark. An important property of this method is that it makes it possible to execute it in interval analysis and as a consequence allows us to obtain reliable error bound by using Lagrange's error formula for Taylor series[4].

In fact, this example illustrates the method known as *Forward accumulation*, where in order to compute the n-th normalised derivative of the Taylor coefficient, we begin by computing the values for $p = 1, 2, \ldots, n$ in a consecutive order. An alternative method would be to start with the values of n, and traverse backwards to $n = 1$. This is also known as *Backward accumulation*

3.4 Forward and Backward accumulation

In this section we will describe two of the most commonly used procedures for automatic differentiation: Namely, *forward* and *backward accumulation*, also known as Tangent and Adjoint modes respectively.
In simplified terms, difference between forward and backward accumulation boils down to the order in which we multiply derivatives, or Jacobians for multi-variate functions. Essentially, we refer to the order in which the Kantorovich graph of the function is traversed. In the following section,a brief introduction to both methods will be given. For deeper dive on the topic, refer to[9].
In a more general sense, we take the arbitrary function $y = f(g(h(x)))$, where f is the composition of a sequence of once continuously differentiable elemental functions, and the first order derivative of the function can be obtained by the chain rule as:

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial x}$$

As an example, we take the same Bernoulli equation described in (23), which has the following Kantorovich graph:

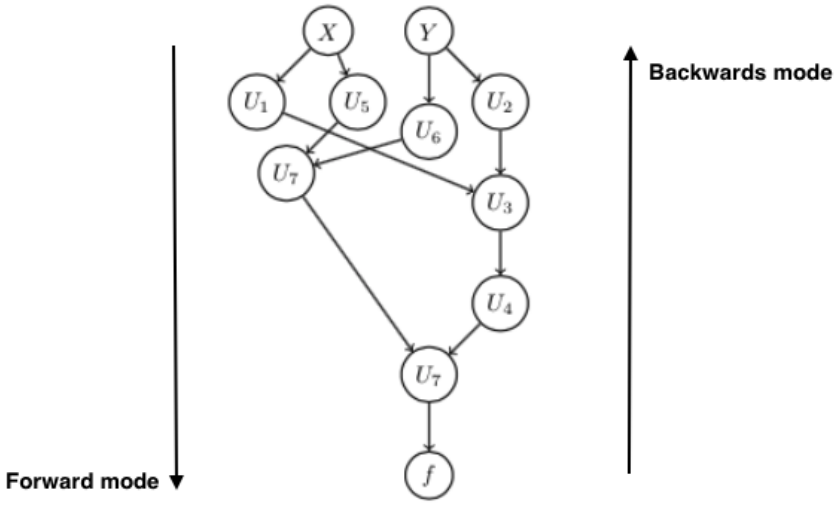


Figure 1: Kantorovich graph of f

The order of evaluating the chain rule is represented by the arrows in Figure 1.

Hence, with forward accumulation, we begin by calculating $\frac{\partial h}{\partial x}$ and then $\frac{\partial g}{\partial h}$ followed by $\frac{\partial f}{\partial g}$ whereas the order is reversed in backward accumulation.

Without loss of generality, the idea above also extends to multivariate functions. Consider

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

Where we would like now to compute all partial derivatives of $f_1, f_2, ..., f_m$ w.r.t. $\boldsymbol{x} = (x_1, x_2, ..., x_n)$
We traverse the graph to obtain the Jacobian of f

$$\nabla f = J_f = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

And then we simply proceed to compute all the derivatives, where for each node on the Kantorovich graph we augment each intermediate variable during evaluation of a function with its derivative and store it so that for u_i we obtain a tuple, containing both itself and its derivative: $u_i \longrightarrow (u_i, \dot{u}_i)$

Since both forward and backward accumulation are popular methods used to differentiate a function with automatic differentiation, it raises the question on how to decide which mode is suitable in practice.
Essentially, a good rule of thumb is to consider a function: $f : \mathbb{R}^n \to \mathbb{R}^m$

- In case the amount of input variables is larger than that of the output variables, i.e. $n \gg m$, backward accumulation would be more suitable. For example, in a number of modern Machine Learning problems where the amount of input data is vast and is larger than the output, like Neural Networks[21]. Hence, an important advantage of the backward mode is that it is significantly less costly to evaluate than the forward mode for functions with a large number of inputs.

- Similarly, in case the number of output variables exceeds the number of input variables i.e. $n \ll m$, then we should use forward accumulation as it would be less costly to evaluate the derivatives than the backward mode.

Remark. Note that in practice, most implementations take this into consideration and have a built-in method to determine which order would be optimal to use, so choosing between the two methods becomes less of a consideration. Some Taylor method implementations using automatic differentiation can be found in [4, 5, 7].

4 Numerical Simulations

There is a need for highly accurate numerical integrators, for example in some problems of Mechanics and Dynamical Systems - For instance, problems in celestial mechanics and molecular dynamics that can be very slow for computations that need extended precision arithmetic[7]. Therefore, we would like to examine whether a Taylor integrator would be suitable for such problems. In order to increase the accuracy of the numerical integrator, the Taylor method does not need to reduce the step size h ; It is possible to increase the accuracy by simply increasing the order p, which will require computing additional higher order derivatives.

Next to speed and accuracy, Discrete gradient schemes such as the Taylor method, preserve exactly (up to round-off errors) the total energy, as was previously shown in section 2.25 based on the work of [19]. This makes such schemes appealing, in particular for problems such as the N-body system of classical mechanics. Therefore, in our simulation, we will check how does the Taylor method perform on a number of classical problems from Newtonian mechanics: A simple mathematical Pendulum, followed by a double pendulum which will exhibit a chaotic behaviour, and the 2-body problem(Kepler).

4.1 Mathematical Pendulum

A pendulum is a body suspended from a fixed support so that it swings freely back and forth under the influence of a certain force. We denote by θ the angle of displacement and by ω the angular momentum, which are sometimes denoted by x and p respectively. The variables in the equation are: l, which represents the length of the rod; m, which represent the mass, and g, which is a gravitational constant.

In our simulation, we will be using a simple mathematical pendulum, sometimes referred to as a simple gravity pendulum and which was previously formulated in Example 2.4. This is a 2nd order differential equation given by:

$$m\theta'' + \frac{mg}{l}\sin\theta = 0$$

where we pick unit mass m, a mass-less rod l of length 1, and gravitational acceleration $g = 1$. Then, the corresponding equations of motion are:

$$\begin{cases} \theta' = \omega \\ \omega' = -\sin\theta \end{cases}$$

Furthermore, the pendulum can be described by the following Hamiltonian:

$$H(\theta, \omega) = \frac{1}{2}\omega^2 - \cos\theta$$

Which describes the total energy of the system.

Remark. Note that this pendulum has a period of 2π, so in our simulation we will be using a time interval which is larger than that.

4.2 Double pendulum

Remark. Note that in the following two subsections we have chosen to use Newton's notation for differentiation using dots in place of Lagrange's notation of using prime symbols, due to convenience as some of the terms were squared. Newton's notation denotes the derivative of y with respect to the dependent variable x by $\dot{y}$.

A double pendulum, is a pendulum which has an additional pendulum attached to its end. This make it an interesting example to consider, because it exhibits chaotic behaviour and has a strong sensitivity to initial conditions. For convenience, we will use angles, denoted by θ_1 and θ_2, which are the angles between each of the limbs and verticals of the pendulums, as the generalized coordinates defining the configuration of the system. In a similar manner to the simple pendulum, here l_1, l_2 will represents the length of the rods; m_1, m_2 will represent the masses, and g is a gravitational constant. The potential energy of the system is given by:

$$V = -(m_1 + m_2)l_1 g\cos\theta_1 - m_2 l_2 \cos\theta_2$$

And the kinetic energy is given by:

$$T = \frac{1}{2}m_1 l_1^2\dot{\theta}_1^2 + \frac{1}{2}m_2[l_1^2\dot{\theta}_1^2 + l_2^2\dot{\theta}_2^2 + 2l_1 l_2\dot{\theta}_1\dot{\theta}_2\cos(\theta_1 - \theta_2)]$$

We let $\dot{\theta_1} = \omega_1$ and $\dot{\theta_2} = \omega_2$ denote the velocity components.
Therefore, the total energy of the system, is described by the following Hamiltonian:

$$\begin{aligned} H(\theta_1, \theta_2, \omega_1, \omega_2) &= V + T \\ &= -(m_1 + m_2)l_1 g\cos\theta_1 - m_2 l_2\cos\theta_2 \\ &\quad + \frac{1}{2}m_1 l_1^2\omega_1^2 + \frac{1}{2}m_2[l_1^2\omega_1^2 + l_2^2\omega_2^2 + 2l_1 l_2\omega_1\omega_2\cos(\theta_1 - \theta_2)] \end{aligned}$$

Then, the corresponding equations of motion are :

$$\begin{cases} \dot{\theta_1} = \omega_1 \\ \dot{\theta_2} = \omega_2 \\ \dot{\omega_1} = \frac{-g(2m_1+m_2)\sin(\theta_1) - m_2 g\sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1-\theta_2)m_2(\omega_2^2 L_2 + \omega_1^2 L_1\cos(\theta_1-\theta_2))}{L_1(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \\ \dot{\omega_1} = \frac{2\sin(\theta_1-\theta_2)(\omega_1^2 L_1(m_1+m_2) + g(m_1+m_2)\cos(\theta_1) + \omega_2^2 L_2 m_2\cos(\theta_1-\theta_2))}{L_2(2m_1+m_2-m_2\cos(2\theta_1-2\theta_2))} \end{cases}$$

4.3 Kepler's problem

For computing the motion of a planet around the sun, we pick the sun as the centre of our coordinate system and the orbiting body as the planet;
The motion will then stay in a plane and we can use two-dimensional coordinates $\mathbf{q} = (q_1, q_2)$ for the position of the orbiting planet.

Hence, let q_1 and q_2 denote rectangular coordinates centered at the sun, specifying at time t the position of the planet such that the planet moves in elliptic orbits with the sun at one of the focal points. Additionally let $\mathbf{p} = (p_1, p_2)$ denote components of velocity in the q_1 and q_2 directions.
Then, Newton's laws, with a suitable normalization, yield the following differential equations

$$\begin{cases} \ddot{q_1} = -\frac{q_1}{(q_1^2+q_1^2)^{(\frac{3}{2})}} \\ \ddot{q_2} = -\frac{q_2}{(q_1^2+q_1^2)^{(\frac{3}{2})}} \end{cases}$$

Which is equivalent to the following Hamiltonian system:

$$H(p_1, p_2, q_1, q_2) = \frac{1}{2}(p_1^2 + p_1^2) - \frac{1}{\sqrt{q_1^2 + q_1^2}}$$

describing the total energy of the system. Additionally, we let $\mathbf{p} = \dot{\mathbf{q}}$, so that the corresponding equations of motion are:

$$\begin{cases} \dot{q_1} = p_1 \\ \dot{q_2} = p_2 \\ \dot{p_1} = -\frac{q_1}{(q_1^2+q_1^2)^{\frac{3}{2}}} \\ \dot{p_2} = -\frac{q_2}{(q_1^2+q_1^2)^{\frac{3}{2}}} \end{cases}$$

4.4 Methods used

In our implementation, we will be using Heyoka: A modern and general-purpose implementation of Taylor's integration method for the numerical solution of ordinary differential equations developed by Biscani And Izzo[5]. In their paper, they show how Heyoka, as a general-purpose integrator, is competitive with and often superior to state-of-the-art specialised symplectic and non-symplectic integrators in both speed and accuracy.
In order to choose the optimal step size and order for the Taylor method, Heyoka follows the propositions which were first formulated by Jorba And Zou in [4], and are outlined in Section 2.1. There, it is shown how the optimal order is chosen once the required error is specified by the user. For a deeper dive into how the package works, refer to [5].
One of the main advantages of the Heyoka package is that it is usable directly from the Python and C++ lanaguages, and it bypasses the need to adapt the Automatic differentiation integrator for each specific function in order to produce the code list. It does not require the usual intermediate translation step that is found in other such methods in order to adapt the integrator, relying instead on a just-in-time compilation approach.
Furthermore, another advantage of the Heyoka package is that it includes support for extended precision arithmetic; where the user of the package can specify the required error tolerance of the model and then the optimal value for the order of differentiation p will be chosen such that the required accuracy is obtained and the number of operations is minimised. This differs slightly to the approach taken by Jorba and Zou, where one needs to specify both a relative error and an absolute error.
Furthermore, we would compare the result of the Taylor method against the *Leapfrog* model, which is a symplectic integrator, and as such would be a good benchmark to compare the Taylor method against.
Additionally, a non-symplectic, standard Runge-Kutta 4 method, implemented from the matplotlib Python Library will be used. We will primarily be looking to contrast the result of the Taylor and Leapfrog methods to the Runge-Kutta 4 model, as the latter is expected to produce results that deviate from the expected solution of the 2 body

problem for large time steps, since it is not energy preserving. And finally, an additional model will be used to verify our results - Taken from the Scipy.integrate package with the built-in function odeint, using an LSODA adaptive integrator; This would provide a good benchmark to compare the output from the Taylor, Leapfrog and RK4 models.

In order to make the comparison between the models fair, we have kept the same step-size h and time-interval for all models. Additionally, whenever we could indicate the error tolerance for a model we have chosen to use 10^{-15}. This was only possible for the Taylor model and the LSODA adaptive integrator. Note that for the Taylor model, indicating an error tolerance does not have an effect on the step-size h. We are able to minimise the error of the model by increasing the order p, as was shown in Section 2.1, thus requiring higher-order derivatives which would increase the computational complexity. However, with Automatic differentiation, we expect that the Taylor model would produce accurate results while remaining competitive in terms of time and computational effort. As an example, with the previously specified error tolerance,an order of $p = 19$ was obtained in a matter of seconds for all the models specified previously; i.e., the function was differentiated 19 times. If a lower error tolerance would have been specified, the order would have been smaller.

5 Results

5.1 Pendulum

In the following subsection, we would like to see how do the three additional models that were described in section 4.4 compare to the Taylor method for computing an approximation of the solution of the pendulum described in Section 4. For that, we take a sufficiently long time-interval and run a simulation of all four models with varying time-steps for comparision.

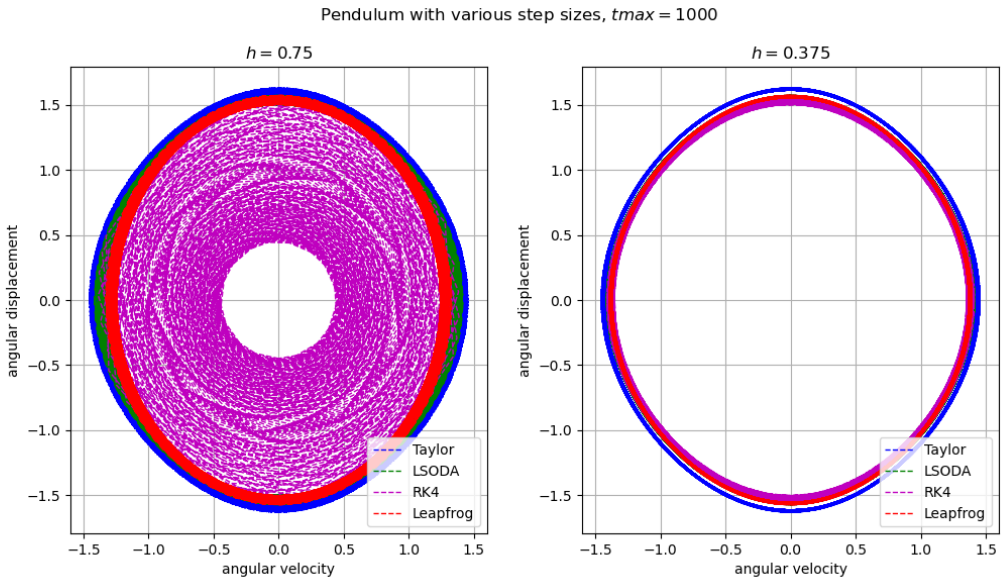


Figure 2: Pendulum. Left:$h = 0.75$, Right:$h = 0.375$, $t_{max} = 1000$

As can be seen in Figure (2), the plots shows the result of each model with varying time-steps. Here, we plotted the angular velocity against the angular displacement of the pendulum. On the right, the models use a small time step, given by $h = 0.375$. There, all models do relatively well and are staying along the path of the orbit. However, when we double the the step size to $h = 0.75$(On the left), which would result in a total of 1333 steps on the interval, the performance of all models suffer in comparison to the previous simulation, which is expected. However, the Taylor, Leapfrog and LSODA models do stay relatively close to the orbit, while the Runge-Kutta model deviates significantly. Additionally, we computed the energy of the system for each model: As can be seen from Figure 3, the energy of the system is preserved within a reasonable range for the Taylor, Leapfrog and LSODA models, while it is decreasing for the Runge-Kutta model.

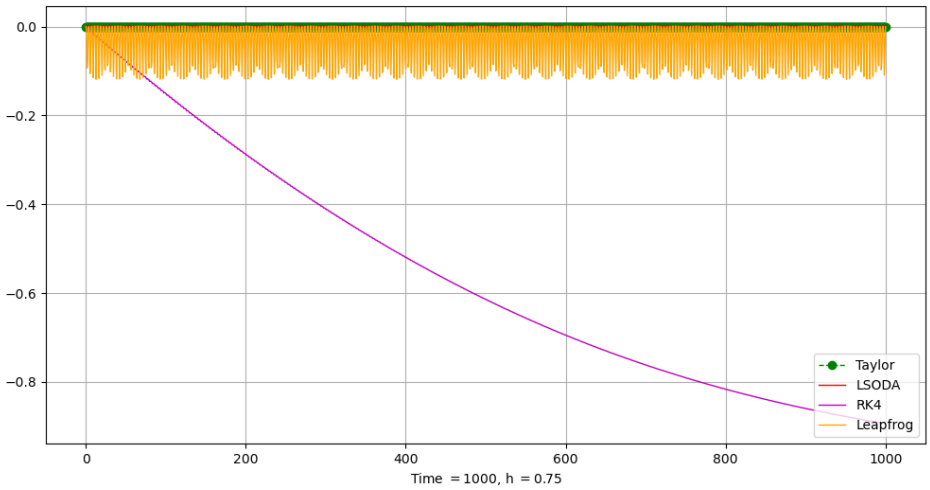


Figure 3: Energy of the pendulum for each model

5.2 Double-pendulum

For the double-pendulum, we used the equations of motion formulated previously in Section 4. Here, we let $l_1 = l_2 = m_1 = m_2 = 1$ and $g = 9.81$. Furthermore, since we use a bigger gravitational constant than for the simple pendulum, the frequency of the pendulum will increase so that we choose a smaller time-interval of $[0, 250]$. Additionally, we repeated the simulation for various values of h for comparison.
We began by simulating the Taylor, LSODA, Leapfrog and Runge-Kutta models.

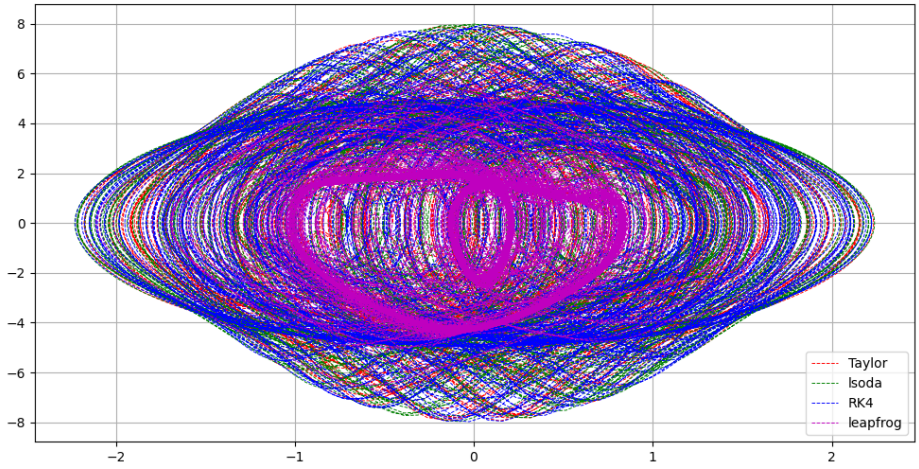


Figure 4: Double-Pendulum, $h = 0.05$

Figure 4 shows a phase-space diagram of the angular velocity against the angular displacement of one of the pendulums, and exhibits chaotic behaviour.
Furthermore, we continued with the comparison by computing the energy of each model:

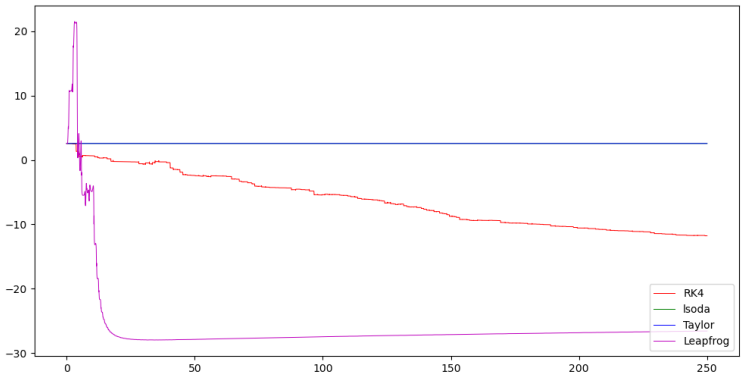


Figure 5: Energy, $h = 0.05$

In Figure 5, we can see a time series of the total energy of the system with all models for $h = 0.05$. As can be seen, the Taylor and LSODA integrators perform particularly well in terms of preserving the energy. In fact, upon closer inspection, the energy of the system using the Taylor integrator stays virtually fixed on a straight-line. This agrees with the result of Theorem 2.25. In contrast, it can be seen that the energy of the system is decreasing for the Runge-Kutta integrator, and that the energy from the leapfrog model is not preserved.

We note that the order of the Leapfrog and Runga-Kutta models is of order 2 and 4 respectively, while the other models are of a much higher order. Therefore, we will try to repeat the simulation for a smaller step-size in order to see whether that improves the performance of the models. Letting $h = 0.01$, we increase the number of steps by 5, while keeping the same time-interval. Repeating the simulation, we obtained the following graph:

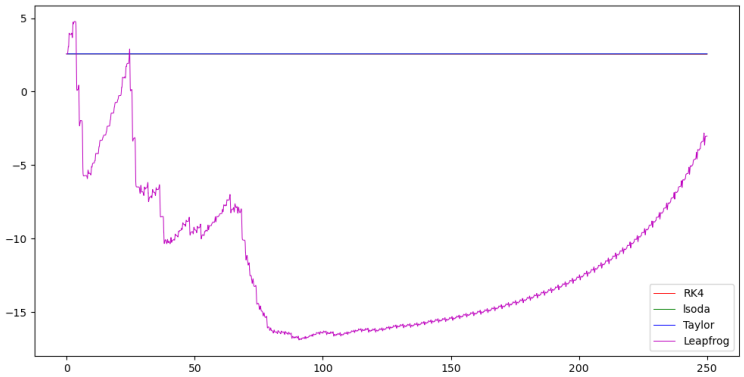


Figure 6: Energy of the double pendulum, $h = 0.01$

As can be seen from Figure 6, the total energy of the system for the Taylor, LSODA and Runge-Kutta models is preserved. However, while the bound of the error for the Leapfrog model was reduced, the energy was still not preserved. Therefore, reducing the step-size did enhance the accuracy of all models, although not equally.

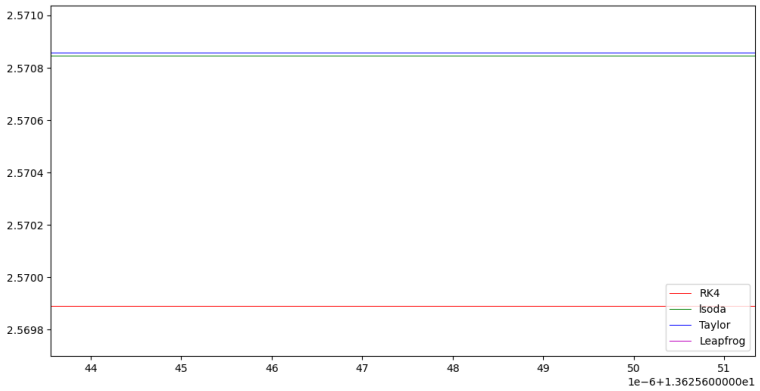


Figure 7: Zoomed in : Energy of the double pendulum, $h = 0.01$

Additionally, by zooming in on Figure 6, we are able to have a closer look on the energy of the models, which is illustrated in Figure 7. There, we can see that the differences between the Taylor, LSODA and Runge-Kutta is indeed very small.

Since the double-pendulum is a chaotic system, this could be a case where the Leapfrog model does not perform well enough with large time-steps, due to the fact that the model is only of order 2. Thus, in cases where the error is not small enough, the model jumps to different behaviours and loses the energy-conservation property; i.e. energy remains bounded but is not well conserved. In fact, only by reducing the step-size and interval significantly to $t_{max} = 25$ and $h = 0.0001$ we were able to achieve reasonable conservation of energy using the Leapfrog model.

5.3 Kepler

In our simulation of the Kepler problem from Section 4, we used a time interval of $[0, 1000]$ and the following initial conditions:

$$q_1(0) = 1 - e, \;\; q_2(0) = 0, \;\; \dot{q_1}(0) = 0, , \;\; \dot{q_2}(0) = \sqrt{\frac{1+e}{1-e}}$$

Where e denotes the orbital eccentricity of the resulting orbit, and we use $e = 0.206$ The result of our simulation is as follows:

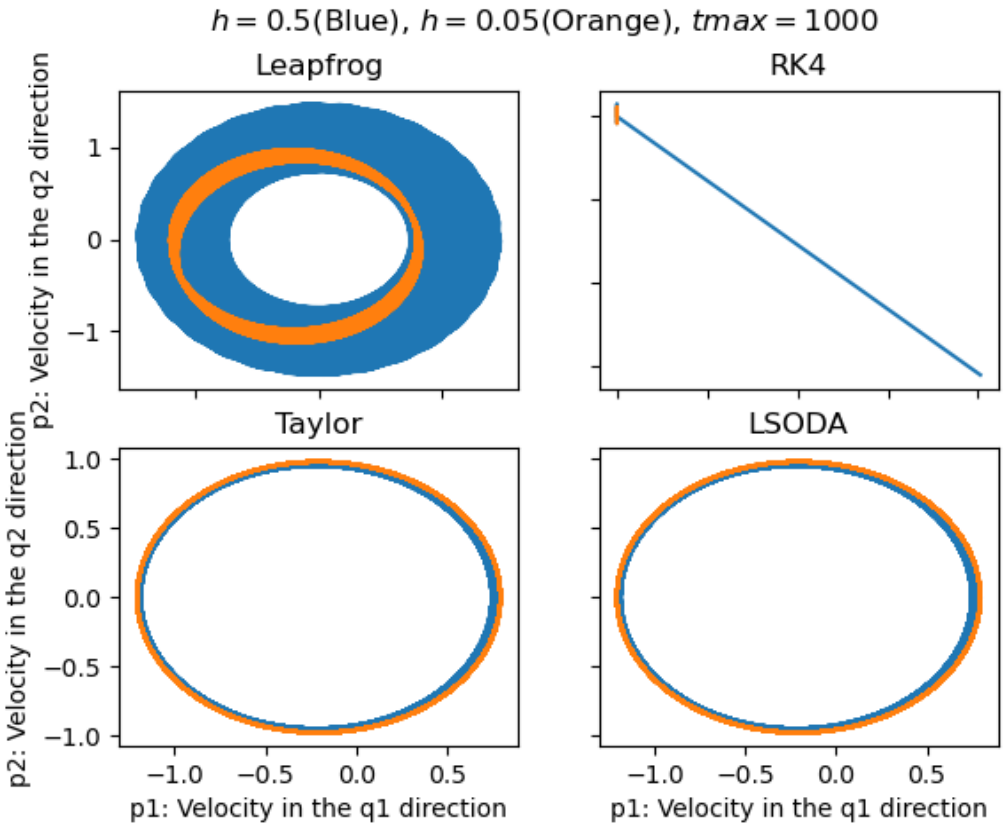
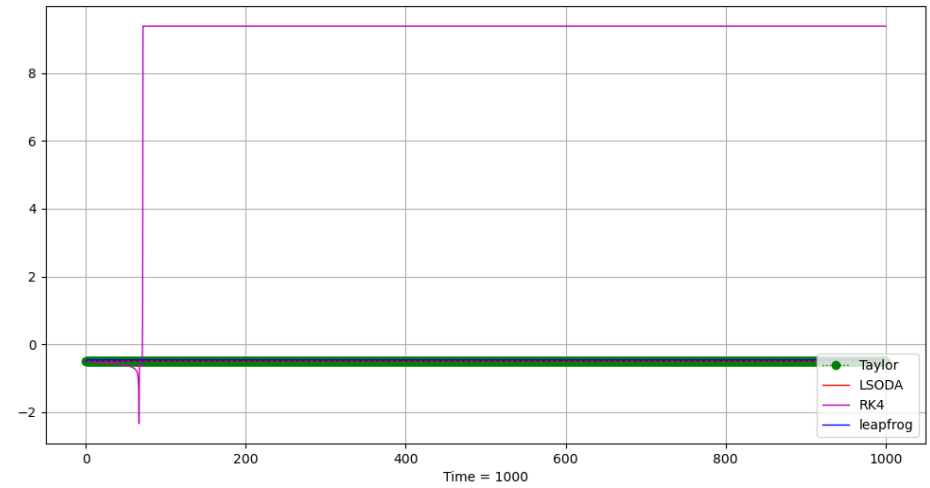


Figure 8: Kepler's orbit for $h = 0.5$(Blue), $h = 0.05$(Orange) and $t_{max} = 1000$

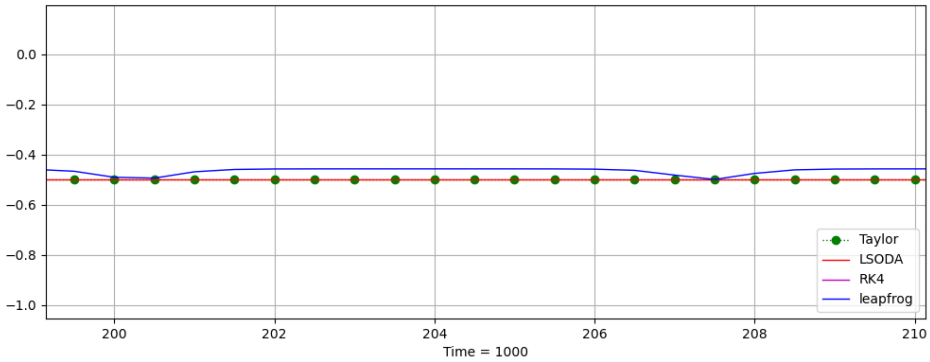
As can be seen from Figure (8), for the small time step given by $h = 0.05$ all models perform sufficiently well, and are orbiting around the singularity. However, when we increase the step-size by a factor of 10 to $h = 0.5$, the orbit of the Runge-Kutta model blows up, while the orbits of the three other models stay within the range.

Furthermore, we calculated the energy of each model along the time interval. Note that the energy of of the Kepler problem is constant and is equal to $-\frac{1}{2}$.

As can be seen from Figure 9, the Taylor and LSODA models do extremely well in preserving the energy. The energy of the system from the Leapfrog model is also preserved, but deviates ever-so-slightly from the Taylor method upon closer inspection. However the energy of the RK4 model is decreasing. This is expected, and is similar to the result



(a) Energy Kepler



(b) zoomed in

Figure 9: Energy of Kepler

which was previously obtained with the double pendulum using the Runge-Kutta model. However, we note that for the Kepler problem, the Leapfrog model does not have the same issues as was the case for the Double-pendulum, and is performing quite well.

Additionally, in order to further check on the accuracy of the models, we tested the error at a particular point on the grid for two time steps: In particular, the point we were interesting in checking was $t = 21\pi$, where we know that the exact solution of $y(21\pi)$

Remark. 21π was chosen simply because we needed an odd multiple of π, which was further along the time grid. By applying Kepler's third law, the following Keplerian orbit has a period of 2π. Hence, choosing a multiple of π is a convenient way to verify

the results as we know that the exact solution is given by:

$$y((2k+1)\pi) = \Big[-1-e, 0, 0, \sqrt{\frac{1-e}{1+e}} \Big]$$

Note that given the nature of π, i.e. an irrational number, we were never able to land exactly on the point of 21π. However, we expect the models to have sufficiently close results as we decrease the time-step.

For $h = 0.05$ we obtained the following errors:

Taylor	8.30490645e-07	1.26114391e-03	1.06864508e-03	5.58773955e-07
LSODA	4.01720340e-07	1.19183772e-03	1.01071852e-03	2.60623304e-07
RK4	8.30580229e-07	1.26114771e-03	1.06864747e-03	5.58701370e-07
Leapfrog	2.40174701e-05	1.53955323e-04	2.30366973e-04	1.61881627e-05

While for $h = 0.5$, we found the following errors:

Taylor	0.0002424	0.02154478	0.01825697	0.0001631
LSODA	0.00024053	0.02147565	0.01819917	0.00016129
RK4	0.00043122	0.02457391	0.02069518	0.00014852
Leapfrog	0.06679163	0.41235808	0.30717289	0.06361461

As can be expected, by reducing the step size the accuracy of all the models increase. However, for a larger step size, the Taylor model does comparatively well, especially when put against the Leapfrog model. Note that the Leapfrog model is just a second order integrator so this result is expected.

6 Conclusions

In this thesis, we explained the workings of the Taylor method for numerically integrating ODEs. We showed how, with the help of Automatic differentiation, obtaining derivatives of a function can be done in a computationally efficient way, and thus make the Taylor method competitive with traditional models such as Runge-Kutta. Furthermore, we run a simulation on three problems from Newtonian mechanics, using the Taylor method, along with Leapfrog, Runge-Kutta 4 and LSODA models for comparison. In all problems, the Taylor method did particularly well and preserved the energy of the systems. This is in contrast to the Runge-Kutta method, which blew up for problems that contained a singularity when using a large step size or a small time-interval. When compared to the Leapfrog model, the Taylor model was again very powerful. For the Double-pendulum, the Taylor model managed to conserve the energy of the system even for large step-sizes - In contrast to the Leapfrog model which required a significantly smaller step-size in order to conserve the energy of the system. Additionally, in section 2.1 we have shown how the accuracy of the Taylor method can be improved by increasing the order p while keeping h the same. This makes the method extremely competitive for fields that require highly accurate numerical integrators.

References

[1] Ramon E. Moore. Methods and applications of interval analysis. 1979.

[2] T Fuller. Error in Digital Computation, Volume 1. *The Computer Journal*, 1:188, 1966.

[3] Kenneth Berryman, Richard Stanford, and Peter Breckheimer. The atomft integrator - using taylor series to solve ordinary differential equations. *Astrodynamics Conference*, 1988.

[4] Àngel Jorba and Maorong Zou. A software package for the numerical integration of odes by means of high-order taylor methods. *Experimental Mathematics*, 14(1):99–117, 2005.

[5] Francesco Biscani and Dario Izzo. Revisiting high-order Taylor methods for astrodynamics and celestial mechanics. *Monthly Notices of the Royal Astronomical Society*, 504(2):2614–2628, 04 2021.

[6] Alberto Abad, Roberto Barrio, Fernando Blesa, and Marcos Rodríguez. Algorithm 924. *ACM Transactions on Mathematical Software*, 39(1):1–28, 2012.

[7] Petr Veigend, Gabriela Nečasová, and Václav Šátek. Taylor series based numerical integration method. *Open Computer Science*, 11(1):60–69, 2020.

[8] Roberto Barrio, Fernando Blesa, and Sergio Serrano. Qualitative analysis of the rössler equations: Bifurcations of limit cycles and chaotic attractors. *Physica D: Nonlinear Phenomena*, 238(13):1087–1100, 2009.

[9] Andreas Griewank. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. 2000.

[10] Using Eigenvalues and Eigenvectors to Find Stability and Solve ODEs, 3 2021. [Online; accessed 2021-07-19].

[11] Peter J. Olver. Nonlinear ordinary differential equations, 2017.

[12] Taylor's theorem University of Toronto. url: http://www.math.toronto.edu/courses/mat237y1/20199/notes/chapter2/s2.6.html.

[13] George Corliss and Y. F. Chang. Solving ordinary differential equations using taylor series. *ACM Transactions on Mathematical Software*, 8(2):114–144, 1982.

[14] George Corliss and David Lowery. Choosing a stepsize for taylor series methods for solving odes. *Journal of Computational and Applied Mathematics*, 3(4):251–256, 1977.

[15] E. E. Hairer, S. P. Nørsett, and G. Wanner. Solving ordinary differential equations i. nonstiff problems. *Mathematics and Computers in Simulation*, 29(5):447, 1987.

[16] B.Krauskopf H.W.Broer and G.Vegter. *Global Analysis of Dynamical Systems: Festschrift dedicated to Floris Takens for his 60th birthday*. CRC Press, 1 edition.

[17] C.Lubich E.Hairer and G.Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations (Springer Series in Computational Mathematics, 31)*. Springer, 2nd ed. 2006. 2nd printing 2010 edition.

[18] G R W Quispel and D I Mclaren. A new class of energy-preserving numerical integration methods. *Journal of Physics A: Mathematical and Theoretical*, 41(4):045206, 2008.

[19] JL.Cieśliński and B.Ratkiewicz. Energy-preserving numerical schemes of high accuracy for one-dimensional hamiltonian systems. *Journal of Physics A: Mathematical and Theoretical*, 44(15):155206.

[20] Louis B. Rall. Automatic differentiation: Techniques and applications. *Lecture Notes in Computer Science*, 1981.

[21] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey, 2018.

7 Appendix

7.1 Code

7.1.1 Functions

```
#Importing all the necessary packages
import heyoka as hy
import numpy as np
from numpy import linalg as LA
from matplotlib.pylab import plt
from scipy.integrate import odeint

def pendulum(theta,t):
    """simple pendulum x' =v and v' = - sin x"""
    dtheta1_dt = theta[1]
    #dtheta2_dt = -9.81*np.sin(theta[0]) #Angular displacement
    dtheta2_dt = -1 *np.sin(theta[0])
    dtheta_dt = [dtheta1_dt, dtheta2_dt] #Angular velocity
    return dtheta_dt

def double_pendulum(y, t):
    """Double pendulum function: Returns the first derivatives of y = theta1,
        z1, theta2, z2."""
    theta1, z1, theta2, z2 = y
    c, s = np.cos(theta1-theta2), np.sin(theta1-theta2)
    theta1dot = z1
    z1dot = (1*9.81*np.sin(theta2)*c - 1*s*(1*z1**2*c + 1*z2**2) -
            (1+1)*9.81*np.sin(theta1)) / 1 / (1 + 1*s**2)
    theta2dot = z2
    z2dot = ((1+1)*(1*z1**2*s - 9.81*np.sin(theta2) + 9.81*np.sin(theta1)*c) +
            1*1*z2**2*s*c) / 1 / (1 + 1*s**2)
    return theta1dot, z1dot, theta2dot, z2dot

def kepler(x,t):         # Kepler
    p = x[len(x)//2:]
    q =x[:len(x)//2]
    dp1 = p[0]
    dp2 = p[1]
    dq1 = -q[0]/(LA.norm(q, ord=2))**(3)
    dq2 = -q[1]/(LA.norm(q, ord=2))**(3)
    d = np.array([dp1, dp2, dq1, dq2])
    return d

def energy1(y):
    """Return the total energy of the double pendulum."""
    th1, th1d, th2, th2d = y.T
    V = -(1+1)*1*9.81*np.cos(th1) - 1*1*9.81*np.cos(th2)
    T = 0.5*1*(1*th1d)**2 + 0.5*1*((1*th1d)**2 + (1*th2d)**2 +
           2*1*1*th1d*th2d*np.cos(th1-th2))
    return T + V

def energy2(z):
    """Return the total energy of kepler."""
    q1,q2,p1,p2 = z.T
    T = 0.5*(p1**2 + p2**2)
```

```
    V = -(q1**2 +q2**2)**(-0.5)
    E = T+V
    return E
```

7.1.2 Pendulum

```
init = [0, np.pi / 2]           #Initial conditions
# Time interval, amount of steps and step size
tmax, h = 1000, 0.25
t = np.arange(0, tmax+h, h)

#Heyoka
x, v = hy.make_vars("x", "v")  # Create the symbolic variables x and v.

ode_sys = [(x, v), (v, -hy.par[0] * hy.sin(x))]              #Heyoka

ta = hy.taylor_adaptive(ode_sys,theta_0,pars=[1]) # Create the integrator
    object for Heyoka.

#Solutions
status, min_h, max_h, nsteps,sol1 = ta.propagate_grid(t)         #Heyoka
    solution
sol2 = odeint(pendulum, init, t)                          #SciPy OdeInt
sol3 = rk4(pendulum, init, t)                             #RK4
```

7.1.3 Double pendulum

```
# rod lengths, masses, gravitational constant
L1, L2,m1, m2 ,g = 1,1,1,1,9.81
# Time interval, amount of steps and step size
tmax, h = 30, 0.01
t = np.arange(0, tmax+h, h)
# Initial conditions: theta1, dtheta1/dt, theta2, dtheta2/dt.
init = np.array([3*np.pi/7, 0, 3*np.pi/4, 0])

# Creating the integrator object for Heyoka.
t1, z1, t2, z2= hy.make_vars( "t1", "z1", "t2", "z2")
c, s = hy.cos(t1-t2), hy.sin(t1-t2)
ta = hy.taylor_adaptive(
    # Hamilton's equations.
    [(t1, z1),
     (z1,
         ((-hy.par[0]*(hy.par[5]*hy.par[1]+hy.par[2])*hy.sin(t1)-hy.par[2]*hy.par[0]*hy.sin(t1-
         hy.par[5]*t2) -hy.par[5]*hy.sin(t1-t2) *hy.par[2]*(hy.par[4]*z2**2 +
         hy.par[3]*z1**2 * hy.cos(t1-t2)))
          /(hy.par[3] *hy.par[5]*
              hy.par[1]+hy.par[2]-hy.par[2]*hy.cos(hy.par[5]*t1-hy.par[5]*t2)))
              ),
     (t2, z2),
     (z2, ((hy.par[5]*hy.sin(t1-t2))*(hy.par[3]*z1**2 * (hy.par[1] +hy.par[2])+
         hy.par[0]*(hy.par[1]+hy.par[2])*hy.cos(t1) + hy.par[4]*hy.par[2]*z2**2
         * hy.cos(t1-t2))/
```

```
        (hy.par[4]*(hy.par[5]*hy.par[1]+hy.par[2]-hy.par[2]*hy.cos(hy.par[5]*t1-hy.par[5]*t2))))
            )],
    # Initial conditions.
    [3*np.pi/7, 0, 3*np.pi/4, 0], pars=[9.81,1,1,1,1,2] #[g,m1,m2,L1,L2,
        variable]
)



# Does the numerical integration of the equations of motion
sollsoda = odeint(double_pendulum, init, t)                    #LSODA,
solrk = rk4(double_pendulum, init,t)                    #RK4
_, _, _, nsteps, soltaylor = ta.propagate_grid(t) #Taylor

#Computes the energy
E_lsoda = energy1(sollsoda)
E_rk = energy1(solrk)
E_taylor = energy1(soltaylor)
```

7.1.4 Kepler

```
# Create the symbolic variables.
q1, q2, p1, p2= hy.make_vars( "q1", "q2", "p1", "p2")

e1 = 0.206 # orbital eccentricity
# Time interval, amount of steps and step size
tmax, h = 250, 0.25
t = np.arange(0, tmax+h, h)
# Initial conditions.
ic = [1-e1, 0,0, ((1+e1)/(1-e1))**(0.5)]

# Construct the integrator object.
ta= hy.taylor_adaptive(
    # Hamilton's equations.
    [(q1, p1),
     (q2, p2),
     (p1, -q1/(q1**2 + q2**2)**(3/2)),
     (p2, -q2/(q1**2 + q2**2)**(3/2))],
    # Initial conditions.
    [1-e1,0,0,((1+e1)/(1-e1))**(0.5)])

# Integrate.
status, min_h, max_h, nsteps, soltaylor = ta.propagate_grid(t) #Taylor
solrk = rk4(kepler, ic, t)                                     #RK4
solint = odeint(kepler, ic, t)                                 #LSODA

#Computes the energy
en_tay = energy2(soltaylor)
en_rk = energy2(solrk)
en_lsoda = energy2(solint)
```