Optimising Taylor Model Computations

Alex Eyre

4th Year Project Report
Computer Science and Mathematics
School of Informatics
University of Edinburgh

2024

Abstract

Taylor models (TMs) are pairs consisting of a real-valued polynomial approximation of a function, and a remainder interval indicating the approximation's error in relation to the original function. They can be used to make formal statements about the behaviour of complex systems without having to compute their analytic solutions.

Many common arithmetic operations can be lifted to work on Taylor models directly, such as multiplication or integration, allowing us to perform typical arithmetic whilst maintaining a rigorous over-approximative guarantee.

In this paper we focus on two methods of optimisation: Lazy evaluation and Partial evaluation. These optimisations, when applied to Taylor model arithmetic, can result in over 10x increases in performance. We introduce a Dependency Graph (DG) structure to implement these optimisations in a transparent fashion.

We implement *TaylorFlow*, a Taylor model arithmetic library using this DG approach in `C++`, along with classes for intervals, dense polynomials, sparse polynomials, and Taylor models. We provide a streamlined library interface for constructing with Taylor models and performing Taylor model arithmetic in `C++`.

We then provide benchmarks and examples utilising this library, to demonstrate its correct function and the performance increases the targeted optimisations provide. We provide some limitations of the library, in addition to a few possible areas of further work to improve the library.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Alex Eyre*)

Acknowledgements

I would like to thank my supervisor, Paul Jackson, for his patience and explanations in the face of my incessant questions.

I would also like to thank James, Ramsay, Nicola, and my family, for their feedback, and for listening to me ramble on about Taylor models for the past few months.

Contents

Chapter 1

Introduction

We are perpetually surrounded by complex systems with complex models, such as the way populations of animals interact, the way heat dissipates through materials, or the way a car's suspension oscillates going over a bump. It is often infeasible to compute solutions to these systems directly, and we must turn to approximations in order to have any hope of understanding their behaviour.

1.1 Taylor models

Taylor models (TMs) present a middle ground for computation, between using a direct perfect solution and an approximation on its own, permitting use of an approximation whilst accounting for the error such an approximation introduces. A Taylor model of a function f is notated (p, I), where p is a polynomial approximation of f, and I is an interval indicating the most the approximation differs from the function f on some domain D. This allows us to make rigorous statements about the behaviour of a system, such as whether it avoids a given scenario, without having to compute an analytic solution.

1.2 Optimisations

When doing arithmetic with a Taylor model approximation, we often encounter situations that can be made significantly more efficient by employing certain optimisations.

We focus on two broad categories of optimisation in this paper

- (**Lazy evaluation**) For instance, given a Taylor model (p, I), we often only require the polynomial approximation p, and do not have to calculate its interval component, in this case we are being 'lazy' about calculating I.
- (**Partial evaluation**) If we have a Taylor model (p, I) as an input to some complex algebraic expression, and we modify I, due to the way Taylor model arithmetic is defined we do not necessarily have to reevaluate the whole expression, but rather only a limited number of results. In this case, the algebraic expression is

partially applied, with the only parameter being the changing interval I, re-using previously calculated results on p.

1.3 Project goals

In this project, we aim to:

- Implement data-types for intervals, polynomials, and Taylor models, in order to allow the user to construct systems and their Taylor model approximations.
- Provide lazy and partial evaluation for Taylor model arithmetic in as transparent a way as possible.
- Implement some common Taylor model arithmetic scenarios, such as Picard iteration, and benchmark them to demonstrate the efficacy of these optimisations in relation to Taylor model arithmetic.

These optimisations are not novel in the field of Taylor model arithmetic. For example the Taylor model library included with the tool Flow* [12] utilises both. However, existing implementations either suffer from a lack of adoption or poor implementation. For example, Flow* implements lazy and partial evaluation via a structure called a `RangeTree`, but this structure must be constructed and managed by the user. Furthermore, Flow* suffers from poor programming practice, for example presenting 26 almost identical, slightly differently optimised functions on its `TaylorModel` class to perform Picard iteration. This makes code written using the Flow* Taylor model arithmetic library hard to understand, and results in difficultly telling if the code is optimised at all.

Therefore, it is a key goal of this project to take as much of the work off of the user as is possible to achieve the two optimisations, ideally making them entirely transparent.

1.4 Contributions

We introduce and implement a Dependency Graph (DG) structure, with two types of vertices: concrete vertices; and deferred computational vertices. Concrete vertices contain values, such as a polynomial or an interval. Deferred computation vertices contain expressions in terms of other vertices to produce some output. Edges on the graph represent 'usage' of a result of one vertex by another, or a *dependency*.

We then utilise the relationship information captured by this graph to obtain both lazy and partial evaluation, via propagation of values and updates as minimally as possible. We implement in `C++`: the graph structure; classes to represent polynomials (univariate and multivariate) and intervals; and provide a parser to create multivariate polynomials and intervals from strings in the syntax Flow* uses in its model files.

We also introduce a user-facing set of functions to interact with a DG structure, both to add concrete values and to create deferred computational vertices. This approach reduces the amount of user involvement in the management of the underlying structures

as much as possible, and provides a clean interface for doing optimised Taylor model arithmetic.

We then apply the library to some illustrative and common scenarios of Taylor model arithmetic, and demonstrate the performance savings their use provides.

1.5 Outline

The report is structured as follows:

- **Chapter 2 - Background & Motivation**: We define Taylor models and their arithmetic, and introduce the targeted optimisation techniques and their application to Taylor model arithmetic.
- **Chapter 3 - Design**: We outline the Dependency Graph structure, and explain how it can be used to implement both Lazy and Partial evaluation. We then discuss representations of the required algebraic data-types to represent a Taylor model, in addition to a Taylor model itself.
- **Chapter 4 - Implementation**: We discuss the process of implementing the design in `C++`, and the adjustments required to create efficient, usable code.
- **Chapter 5 - Evaluation**: We evaluate the project against the project goals, and benchmark the library against itself with and without the targeted optimisations, in order to demonstrate their benefit.
- **Chapter 6 - Conclusion**: We summarise the work done, and discuss limitations and potential for future work.

Chapter 2

Background & motivation

In this chapter we present the background information required to understand the project. Firstly, we introduce Taylor models and briefly outline their applications. Secondly, we motivate why partial and lazy evaluation present an opportunity to improve the performance Taylor model arithmetic. Finally, we examine some existing libraries and their shortcomings, and propose where this project hopes to provide value.

2.1 Taylor models

2.1.1 What is a Taylor model?

Taylor models (TMs) are a mathematical structure consisting of two elements: some real-valued polynomial; and a real-valued closed interval. For instance, take the Taylor model (p, I): the polynomial $p \in \mathbb{R}[x_1, \ldots, x_n]$ can be viewed a function on n variables, and is an approximation to some other function f. The interval component $i = [a, b]$ such that $a, b \in \mathbb{R}$, referred to as the *remainder interval*, represents the error introduced by using the approximation p on some domain $D \subset \mathbb{R}^n$ versus the original function. This interval is always an over-approximation, in that it is guaranteed that any error introduced is captured by the interval, but it is not guaranteed that this interval is necessarily as 'tight' as it can be. We then say that (p, I) is a Taylor model of f on domain D.

Definition 2.1.1. Let $f : D \subset R^v \to R$ be a function that is $(n+1)$ times continuously partially differentiable on an open set containing the domain D. Let c be a point in D and p the n-th order Taylor polynomial of f around c. Let I be the interval such that

$$\forall x \in D, f(x) \in p(x - c) + I \tag{2.1}$$

Then we call the pair (p, I) an n-th order Taylor model of f around c on D [17, pp. 383–384].

For example, we can construct a Taylor model for $f(x) := \sin x$ on the domain $x \in [0, 1/2]$ by computing its Taylor series up to order 5 (where 5 is arbitrarily chosen)

$$p(x) := x - \frac{x^3}{3!} + \frac{x^5}{5!}. \tag{2.2}$$

and then as this is a demonstration of the concept and we have the underlying analytic function, we can simply calculate the remainder interval directly and round to the nearest integer to ensure over-approximation

$$g(x) := \sin x - p(x)$$
$$g([0,1/2]) \subseteq [-1,1].$$

Thus $(p,[-1,1])$ is a Taylor model of sin of order 5 on the domain $[0,1/2]$, note that this remainder interval is *not* as 'tight' as it could be, but that it is a sufficient over-approximation as it encloses the error.

Taylor models were originally introduced by Berz in 1997 in order to formalise approaches to problems encountered in beam physics [3]. This use-case happened to have a much wider set of applications than initially thought, and as such he expanded the ideas again in a follow-up paper with Makino to include verified integration operations [5], a key development for their application to model checking techniques.

2.1.1.1 Interval arithmetic

Interval addition and subtraction work similarly to their typical counterparts, with the upper and lower bounds of both intervals being added or subtracted, respectively.

Definition 2.1.2 (Interval addition and subtraction)**.** Let $a_1, b_1, a_2, b_2 \in \mathbb{R}$ be the bounds of two intervals I_1 and I_2, then

$$I_1 \pm I_2 \equiv [a_1, b_1] \pm [a_2, b_2] := [a_1 \pm a_2, b_1 \pm b_2].$$

The result of interval multiplication needs to be the minimum and maximum the product of a value from each operand could attain.

Definition 2.1.3 (Interval multiplication)**.** Let I_1, I_2 be closed intervals on $\mathbb{R}$, then

$$I_1 \cdot I_2 := \left[\min_{x \in I_1, y \in I_2} xy, \max_{w \in I_1, z \in I_2} wz \right]. \tag{2.3}$$

However, care is required when defining interval exponentiation, as if we simply 'expand' exponentiation as in Equation 2.4, we would expect $[-1,1]^2$ to be $[-1,1]$. However, this is not the case, $[-1,1]^2$ is actually $[0,1]$. Expanding in this way assumes we have n variables as a product, whereas we actually have a single variable as in Definition 2.1.4.

$$[a,b]^n \mapsto \underbrace{[a,b] \times [a,b] \times \cdots \times [a,b]}_{n \text{ times}}. \tag{2.4}$$

Definition 2.1.4 (Interval exponentiation)**.** Let I be a closed interval on $\mathbb{R}$, and $n \in \mathbb{N}$, then I raised to the n^{th} power is defined

$$I^n := \left[\min_{x \in I} x^n, \max_{y \in I} y^n \right].$$

2.1.1.2 Taylor model arithmetic

Many typical arithmetic operations can be lifted to Taylor models, allowing for their manipulation in such a way that preserves the over-approximation guarantee of their interval components. We shall define a subset of these operations on Taylor models, in order to later illustrate the application of optimisation techniques to their computation. All the below definitions are sourced from Makino and Berz (2003) [17].

Definition 2.1.5 (Taylor model addition and subtraction)**.** Let (p_1, I_1) and (p_2, I_2) be Taylor models as defined in Definition 2.1.1, then

$$(p_1, I_1) \pm (p_2, I_2) := (p_1 \pm p_2, I_1 \pm I_2).$$

Definition 2.1.6 (Taylor model multiplication)**.** Let $T_1 = (p_1, I_1)$ and $T_2 = (p_2, I_2)$ be n-th order Taylor models around x_0 over some domain D. Define their product $T_1 \cdot T_2$ by

$$T_1 \cdot T_2 := (p_1 \cdot p_2, I_{1,2}).$$

Where $p_1 \cdot p_2$ is polynomial multiplication up to order n (i.e. terms of order greater than n have been truncated), and $I_{1,2}$ is the combination of four intervals

$$I_{1,2} := B(p_e) + B(p_1) \cdot I_2 + B(p_2) \cdot I_1 + I_1 \cdot I_2.$$

Where p_e is the part of the polynomial $p_1 \cdot p_2$ of order $n+1$ to $2n$, and $B(p)$ denotes an enclosure of p on the domain D, that is *at least* as 'sharp' as the enclosure obtained by direct interval evaluation of $P(x - x_0)$.

Definition 2.1.7 (Taylor model integration/antiderivation)**.** Using notation from Makino and Berz (2003), we can define the integral $\partial_i^{-1}(p, I)$ (in the algebraic sense) of a Taylor model (p, I) of a function f, with respect to variable i of the function f as

$$\partial_i^{-1}(p, I) := \left(\int_0^{x_i} p_{n-1}(x)\, dx_i, (B(P - P_{n-1}) + I) \cdot (b_i - a_i) \right).$$

Or integrating p_{n-1}, the part of p from orders 0 to $n-1$, and then bounding the highest n-th order term(s) into the new remainder bound.

Note that Taylor model integration often refers to the construction of a Taylor model flowpipe of a system, which we discuss later in Section 2.1.3, rather than algebraic integration. However, as the scope of the paper is primarily on Taylor model arithmetic rather than direct flowpipe construction, henceforth we refer to Taylor model 'integration' in the above sense, and make explicit when we refer to Taylor model flowpipe construction.

2.1.2 Picard iteration

Picard iteration is an iterative approach to computing an approximate solution to an Ordinary differential equation (ODE), based on the Picard-Lindelöf theorem [10]. It

works by repeatedly applying the Picard operator, Equation 2.5, where $g_0 := \vec{x}_0$, and $g_{i+1} := \mathbb{P}_F(g_i)(\vec{x}_0, t)$.

$$\mathbb{P}_F(g)(\vec{x}_0, t) := \vec{x}_0 + \int_0^t F(g(\vec{x}_0, s), s)\, ds. \tag{2.5}$$

This operator has wide application to Taylor models, as the error introduced by approximating the solution to the ODE can be captured by the remainder interval of the Taylor model of the approximation. If we desire a k-order TM approximation, we apply the Picard operator in two stages. Firstly, by repeatedly applying the Picard operator k times to the initial conditions $g_0 := \vec{x}_0$, discarding terms with order greater than k. This is guaranteed to converge in at most k iterations for a k-order approximation, by the Picard-Lindelöf theorem [9][10].

For example, if we have a one-dimensional ODE as follows, with an initial set $x_0 \in [0, 1/2]$ and $t \in [0, 1/10]$,

$$F(x) := \frac{dx}{dt} = 1 + x^2. \tag{2.6}$$

Then Picard iteration with $k = 3$ will result in the following approximations g_i, with g_0 being set to the initial conditions x_0, n.b. we omit all but the final value of g_i for brevity,

$$\begin{aligned}
g_0(t) &= x_0 = [0, 0.5] \\
g_1(t) &= x_0 + \int_0^t F(g_0(t), s)\, ds \\
g_2(t) &= x_0 + \int_0^t F(g_1(t), s)\, ds \\
g_3(t) &= x_0 + \int_0^t F(g_2(t), s)\, ds \\
&= [0.333333, 0.729167] * t^3 + [0, 0.625] * t^2 + [1, 1.25] * t^1 + [0, 0.5].
\end{aligned}$$

Secondly, the remainder interval is calculated by taking the now converged k-order approximation g_k, picking an initial 'guess' interval I_0 that is an appropriate enclosure for the polynomial on the domain, i.e. such that $B(F) \subseteq I_0$, and then applying the Picard operator to the Taylor model (g_k, I_0), as in Equation 2.7. This results in an expression in the form $g_k + I_{i+1}$, as we assume that g_k has converged at this point, such that $\mathbb{P}_F(g_k) = g_k$.

$$\mathbb{P}_F(g_k + I_i)(\vec{x}_0, t) = g_k + I_{i+1} \qquad \text{such that } I_{i+1} \subseteq I_i. \tag{2.7}$$

This allows us to compute k-order TM approximations to ODEs, capturing the error introduced by such an approximation, and then to contract the remainder interval to be as tight as possible.

For example, in the case of the system in (2.6), if we perform Picard iteration on the Taylor model (g_3, I_0) where $I_0 := [-1, 1]$ is our initial 'guess' for an appropriate

enclosure of the error of the approximation.

$$\begin{aligned}\mathbb{P}_F(g_3+I_0)(x,t) &= x_0+\int_0^t F((g_3+I_0),s)\,ds\\ &= x_0+\int_0^t 1+(g_3+I_0)^2\,ds\\ &= g_3+\int_0^t 2g_3I_0+I_0^2\,ds.\end{aligned}$$

Evaluating this integral on the domain $x \in [0,1/2], t \in [0,1/10]$ we obtain

$$\mathbb{P}_F(g_3+I_0)(x,t) = g_3+[-0.626396, 0.734356]. \tag{2.8}$$

We can then repeatedly apply the operator to converge to the tightest boundary obtainable via this process, $[-0.572342, 0.626363]$, and therefore we have a Taylor model $(g_3, [-0.572342, 0.626363])$ for the solution to the ODE system F for $x \in [0,1/2], t \in [0,1/10]$.

2.1.3 Flowpipes

As motivation for why these topics prove useful, we introduce the concept of a *Flowpipe*, and explain their application and how Taylor models assist with their computation. Both the below definitions, and the definition of the continuous reachability problem are sourced from Chen (2015).

Definition 2.1.8 (Flowpipe)**.** We define the flow of a continuous ODE system $\dot{x} = f(\vec{x},t)$ as $\varphi_f(\vec{x}_0,t)$, the unique solution of the continuous system with $\vec{x}(0) = \vec{x}_0$ at some time t. Then let the set of flows, or the *flowpipe* on a time interval $[0,\Delta]$ be

$$\varphi_f(X_0,\Delta) := \{\varphi_f(\vec{x}_0,t) | \vec{x}_0 \in X_0, t \in [0,\Delta]\}$$

Where $\dot{x}$ refers to the derivative of x with respect to time, i.e $\frac{dx}{dt}$.

Definition 2.1.9 (Taylor model flowpipe)**.** Given an n-dimensional non-linear continuous system $\mathcal{S} : \dot{\vec{x}} = f(\vec{x},t)$ and an interval or TM initial set $X_0 \subseteq \mathbb{R}^n$, with discrete time-steps $\delta_1,\ldots,\delta_N$, then for $1 \le i \le N$, the i-th TM flowpipe is of the form $\mathcal{F}_i(\vec{x}_0,t) = (p_l(\vec{x}_0,t), I_l)$ such that $\vec{x}_0 \in X_0$ and $t \in [0,\delta_i]$. It is an over-approximation of the flowpipe solution $\varphi_f(\vec{x}_0, \sum_{j=1}^{i-1}\delta_j + t)$, i.e.

$$\varphi_f(\vec{x}_0, \sum_{j=1}^{i-1}\delta_j + t) \in p_i(\vec{x}_0,t) + I_i \quad \text{for all } \vec{x}_0 \in X_0, t \in [0,\delta_i].$$

Informally, when we have a continuous system defined by an ODE $\dot{x} = f(\vec{x},t)$, the path of system starting from the initial condition $\vec{x}_0$ at time $t = 0$ can evolve in various ways over time, each of these possible eventualities could be considered a 'trajectory' of the system. Now, imagine we are not looking at a single trajectory, but all possible trajectories starting from a whole set of initial conditions X_0. The flowpipe over a time interval represents all these possible trajectories, encompassing all states the solutions can reach within the time-frame from any initial condition within X_0. Think of this like

a river: if X_0 is the source, then the flowpipe shows you all the paths the water could take within a certain time-frame.

However, if we do not have an exact solution to the system $\dot{x} = f(\vec{x},t)$, but rather a TM approximation of a solution, we cannot construct the true flowpipe, but rather an over-approximation of the flowpipe based on Taylor model arithmetic, guaranteed to enclose the analytic flowpipe. This means that the actual flow of the system from any point in X_0 within a time interval will lie within the Taylor model flowpipe. Going back to the river analogy, this is like saying 'we cannot trace every droplet of water precisely, but we can tell you the rough area through which it will flow'.

Once we have a Taylor model flowpipe, we can use it to make statements on the set of states the system will not enter given an initial set X_0 in a given time frame. For example, this allows us to make (over-approximative, but useful) statements on the maximum attainable temperature of a drill in a given time frame if we have an ODE system modelling such with inputs such as drill speed, lubricant flow rate, etc. Which is evidently a useful statement for a company designing a drill for oil-extraction.

2.1.3.0.1 Alternatives to Taylor models We do not want to leave the reader with the impression that Taylor models are the only approach to perform this process of validated integration of ODEs. For example one can use interval-based integration methods [20] and avoid Taylor models entirely. This Taylor model approach was pioneered by Berz and Makino [4][3][5][17].

2.2 Optimisation techniques

2.2.1 Lazy evaluation

Lazy evaluation is the idea that we should never compute a result until it is required, or often not at all if it is never used [25]. For example, if we compute the product of two values, but then never access the variable we assign the result to, such as in Listing 2.1, Lazy evaluation means that we would never perform the multiplication.

```
1 x <- 3
2 y <- 7
3 result <- x·y
```

Listing 2.1: Calculation of the value of $3 \cdot 7$, in which the result is never used

More generally, we defer the computation until the point at which it is used, and as a result we defer indefinitely in Listing 2.1, but in Listing 2.2 we defer calculating the value of result until its value is used on line 5 to square it and then pass to some function `print`.

```
1 x <- 3
2 y <- 7
3 result_1 <- x·y
4 // ...
5 print (result)^2 // result calculated at this point
```

Listing 2.2: Deferred calculation of some result until it is passed to a function accessing its value

In the context of Taylor models, which consist of both a polynomial and interval component, we often only require one of these components, and can therefore improve performance by neglecting to calculate the unused portion. For example, if we perform Taylor model multiplication, and never use the interval portion of the result, such as in Listing 2.3, then we do not need to calculate it.

```
1 (p1,I1) <- (1+x^2, [-1,1])
2 (p2,I2) <- (3+y, [0,1/2])
3 (pr,Ir) <- (p1,I1)·(p2,I2)
4 print pr
```

Listing 2.3: Taylor model multiplication, in which only the polynomial portion of the result is used

This situation in which we only require one component of a Taylor model occurs frequently in Taylor model arithmetic, for example if we are using Taylor models for purely polynomial arithmetic in the case of Picard iteration.

2.2.2 Memoisation and Partial evaluation

Memoisation is a specific form of caching in the context of functions. When a function is called, its parameters and resulting value are stored, and if the same function is called with the same parameters, the cached result is retrieved and returned rather than performing the computation again [11]. This can drastically improve performance, especially for expensive functions that are often called with the same arguments.

Compile-time partial evaluation is a well-established technique to create a specialised version of a program at compile-time, given a set of know constants [14]. This is accomplished by viewing a program in terms of data-flow, i.e. as a mapping from an input space to some output space, and then partially applying the program 'function' to produce one of lower arity, which can be more aggressively optimised by the compiler.

In the context of Taylor model arithmetic, although of course memoisation has its place and we should cache results wherever possible. There is space to perform optimisation somewhere between memoisation and compile-time partial evaluation. For instance, taking Taylor model multiplication, this can be viewed as a series of functions to perform the polynomial multiplication, product truncation, polynomial bounding on the domain, and interval multiplication amongst others. One possible expression of the Taylor model product of (p_1,I_1) and (p_2,I_2), with $B(p)$ representing an enclosure of polynomial p on the domain D of the Taylor models, and $p_n, p_{n+1,2n}$ being the polynomial p up to order n and from $n+1$ to $2n$ respectively is:

$$p_{\text{result}}(p_1,p_2) := (p_1 \cdot p_2)_n \tag{2.9}$$

$$I_{\text{result}}(p_1,p_2,I_1,I_2) := B((p_1 \cdot p_2)_{n+1,2n}) + I_1 \cdot B(p_2) + I_2 \cdot B(p_1) + I_1 \cdot I_2. \tag{2.10}$$

With the resulting Taylor model being the results of these two functions, such that $(p_1,I_1)\cdot(p_2,I_2) = (p_{\text{result}}(p_1,p_2), I_{\text{result}}(p_1,p_2,I_1,I_2))$. If we know that only one of these inputs changes, for example I_1 (which for example occurs performing Picard

iteration to contract the remainder interval as in Equation 2.7) we could partially apply the function I_{result}. Many of the calculations are constant in I_1, indicated in green,

$$I_{\text{result},\, p_1,p_2,I_2}(I_1) := B((p_1 \cdot p_2)_{n+1,2n}) + I_1 \cdot B(p_2) + I_2 \cdot B(p_1) + I_1 \cdot I_2. \tag{2.11}$$

If we can partially apply such functions and then re-use the results we know to be constant, via memoisation and capturing each component of an operation in this way, then we can avoid recomputation of results that are unchanging when only a subset of variables change. Although not partial evaluation in the typical, compile-time sense, this approach follows the same general themes of partial evaluation, and the term is often used to refer to this general approach of partial function application [16]. We will refer to this technique as such throughout the paper.

2.3 Existing work

2.3.1 Flow*

Flow* is a verification tool for cyber-physical systems, with a focus on solving the reachability problem [12][9]. It implements Taylor model arithmetic in addition to a number of other features, and allows the user to supply a model file containing an ODE system in addition to an initial set X_0, and additional parameters, and produce a Taylor model flowpipe. Flow* implements optimisations such as lazy evaluation and memoisation via a structure known internally as a `RangeTree` that must be manually passed about, and via a number of similarly named, differently-optimised functions for every operation available under Taylor model arithmetic. This results in a confusing user experience, it is often difficult to tell if a given piece of code utilising Flow*'s Taylor model arithmetic library is optimised, and if so how. This hinders the maintainability of the code, and vastly detracts from the project's ability to grow organically, as the code is poorly organised and implemented from a software engineer perspective.

For example, Flow*'s `TaylorModel` class contains 26 definitions of similarly named functions to perform Picard iteration, all of which have large sections of overlapping logic. This is not conducive to one's understanding of the code-base as a whole, and severely limits its ability to become more widespread. This project attempts to provide value in comparison to Flow* by taking the burden of optimisation off of the user. By implementing these arithmetic operations in a way that they are as transparent as possible to the user, thus increasing productivity whilst using the library, allowing the user to focus on the Taylor model arithmetic they wish to perform.

2.3.2 TaylorModel.jl

`TaylorModel.jl` is a TM arithmetic library [2] written in the Julia language [6]. Choice of language is important here, as although Julia offers advantages in doing scientific computation, and is performant for float-calculations such as these [6]. However it suffers from a lack of adoption [27]. For a software library to succeed it needs to be able to interface with existing libraries, and the choice of `C++` greatly increases this compatibility versus a niche language such as Julia.

Chapter 3

Design

In this chapter, we describe the data-types required to perform Taylor model arithmetic, and the functionality they should have. We introduce the concept of a dependency graph in the context of Taylor model arithmetic, and outline how it can be leveraged to obtain the lazy and partial evaluation optimisations.

3.1 Dependency graph

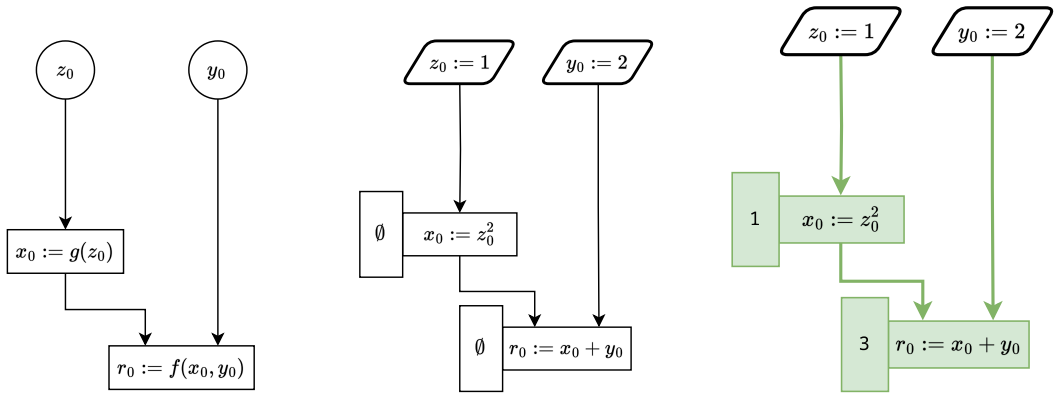
If we have a function, say $f : \mathbb{R}^2 \to \mathbb{R}$, where $f(x,y) := xy$, and we have two values $x_0, y_0 \in \mathbb{R}$, then the function evaluated at this point $r_0 := f(x_0, y_0)$ can be considered to have a 'dependency' on the values of x_0 and y_0. Now suppose that we have another $g : \mathbb{R} \to \mathbb{R}$ and that $x_0 := g(z_0)$ for some $z_0 \in \mathbb{R}$, then $f(x_0, y_0)$ instead now depends on the result of $g(z_0)$ and y_0. This relationship information easily lends itself to a graph structure, for example the above expression $r_0 := f(g(z_0), y_0)$ would look something like Figure 3.1a.

If we now extend this concept Dependency Graph (DG) with different types of vertices, we can use the relationship information it captures to achieve both lazy and partial evaluation.

Vertices on the graph can be one of two things, either they contain a concrete supplied value such as a polynomial or interval, or they contain a function. The inputs of such a function are the values of other vertices and the output of which is a value i.e. a polynomial or interval, and further any vertex containing a function has an associated cache of the function's result. We refer to the vertices containing only a value and no function, and hence have no incoming edges incident on them, as 'concrete vertices', and those those containing a function and cache as 'deferred vertices', represented on the following graphs as $\triangle$s and $\square$s respectively.

3.1.1 Lazy evaluation

With this extended notion of a vertex in hand, lazy evaluation is achieved by simply never calculating the result of an expression if it is never requested. For instance, if we



(a) Dependency graph for squaring and summing two operands

(b) Figure 3.1a augmented with concrete and deferred vertices

(c) Figure 3.1b after the value of r_0 has been requested, new computations and data-flow are shown in green

Figure 3.1: Dependency graphs for $r_0 := z_0^2 + y_0$ at various stages of construction and usage

have the following code to match the previous example, with some arbitrary operations assigned to functions f and g, and some example operands to z_0 and y_0.

```
1 z0 <- 1
2 y0 <- 2
3 x0 <- z0^2
4 r0 <- x0 + y0
```

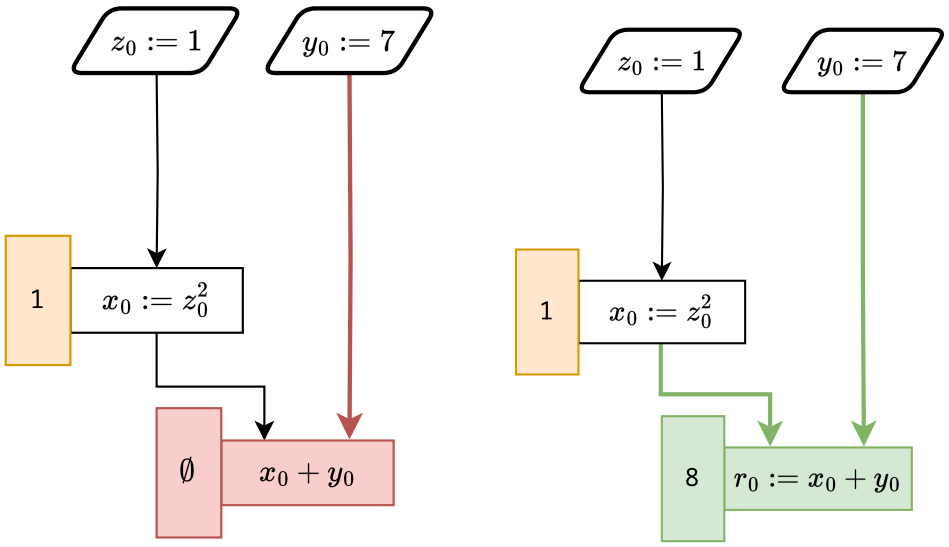
Listing 3.1: Example computation with an unused result r_0

In a computational context, we consider anything observing a result as 'using', for instance printing, doing regular non-deferred arithmetic, etc. In Listing 3.1, we do not 'do anything' with this result r_0, and so if we calculated its value eagerly this would be wasted computation. This small snippet in Listing 3.1 could be represented by the graph in Figure 3.1b, note that we have attached the caches to each computational vertex $\square$, and that they all read $\varnothing$ as they are empty.

Now if we use the value of r_0, we obtain Figure 3.1c. Firstly, r_0 would request the values of all edges incident on it, in this case x_0 and y_0, causing the former to request the values of the vertex with edges incident on it, in this case z_0, before evaluating and caching the result $x_0 := g(z_0)$, and the latter to simply return its value as it is a concrete vertex. Finally, it would compute and cache $r_0 := f(x_0, y_0)$ with these supplied values, and pass it to whatever function requested its value. This results in lazy evaluation, as we track the 'recipe' to compute the value of any computation vertex in the graph, we need not eagerly evaluate each at the time of its declaration, and can instead wait until it is used to do so (possibly forever if it is never used).

3.1.2 Partial evaluation

Partial evaluation is again a traversal of the relations between computations, following outgoing edges of any changed values and performing re-calculation on any encountered



(a) Dependency graph from Figure 3.1c after input y_0 has changed

(b) Figure 3.2a after the result r_0 has been requested

Figure 3.2: Invalidation and recalculation of r_0 given a change in y_0, invalidations are shown in red, retained caches in orange, and new computations and data flow in green

expressions. For example, take the situation we have above in Figure 3.1c, and assume we have at some point previously calculated all the values for some purpose, such as in Listing 3.2.

```
1 z0 <- 1
2 y0 <- 2
3 x0 <- g(z0)
4 r0 <- f(x0,y0)
5 print(r0) // just to use the value of r0
```

Listing 3.2: Example computation with an evaluated result r_0

If we then modify the value of y_0 after we have previously calculated r_0, such as in Listing 3.3, we wish to do the minimum number of computations required to obtain this updated value of r_0

```
1 z0 <- 1
2 y0 <- 2
3 x0 <- g(z0)
4 r0 <- f(x0,y0)
5 print(r0) // just to use the value of r0
6
7 y0 <- 7
8 print(r0)
```

Listing 3.3: Example computation in which the amount of compute required to re-evaluate can be reduced via partial evaluation

We follow the outgoing edges of y_0 to r_0 and invalidate its cache, shown in red in Figure 3.2a, avoiding invalidating the cached value of x_0 as it is unchanged. Then if we request the value of r_0, only one additional computation is performed to compute r_0, and the cached result of x_0 is used, with new computations shown in green in Figure 3.2b.

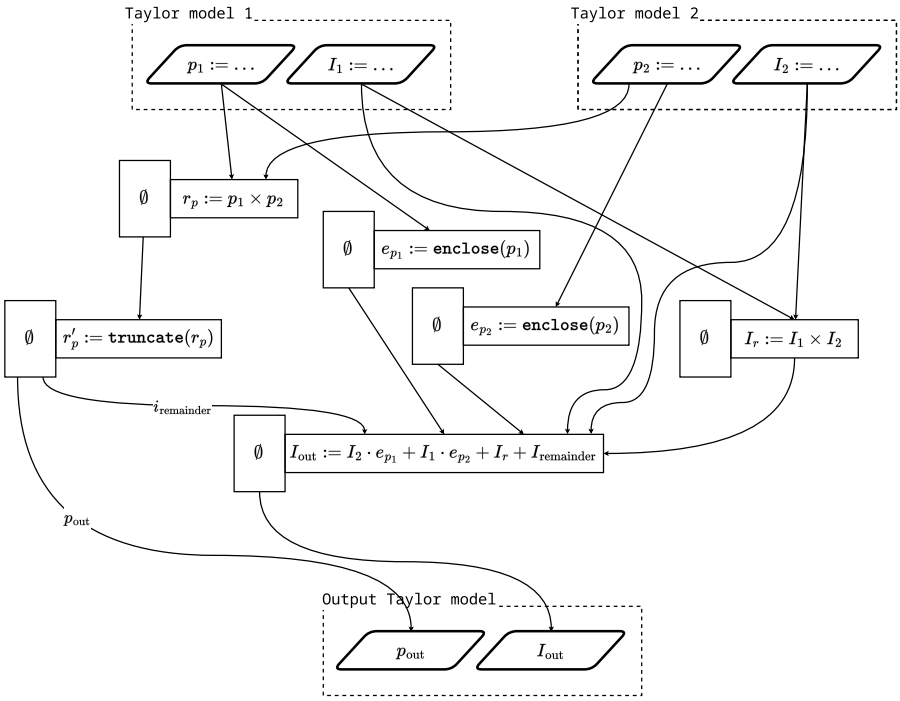


Figure 3.3: Example dependency graph for the multiplication of two Taylor models (p_1, I_1) and (p_2, I_2)

The use of a DG to manage these dependencies masks the underlying complexity of dependency management, as we can present a simplified interface to the user that mimics direct interaction with values. This abstraction layer simplifies user interaction with the library, whilst also ensuring the optimisation of the system via the outlined mechanics. By prioritising the user experience without compromising on computational efficiency, this type of structure supports as wide a user-base as possible by hiding details by default, or exposing them if more fine-tuned tweaking is required.

3.1.3 Worked Taylor model multiplication example

For a more specific example as to why this structure proves useful, consider Taylor model multiplication, which involves performing regular polynomial multiplication and constructing and combining four intervals based on the operands: the interval product of I_1 and I_2; an enclosure for p_1 and p_2 on the domain; and the truncation error introduced when truncating the polynomial product. This results in a dependency graph structure with many vertices as seen in Figure 3.3, where each cache is blank as no computation has been done, and is orange when it contains a cached value.

If we then request only the value of p_{out}, we obtain Figure 3.4. The computations propagate 'up' the tree only to the computations required to obtain this value, shown in green, avoiding computing the interval component of the Taylor model entirely. This is illustrative of how the graph structure allows for lazy evaluation, as although the graph structure will be constructed, computational vertices will only be evaluated if their results are used, and thus we avoid producing unneeded results.

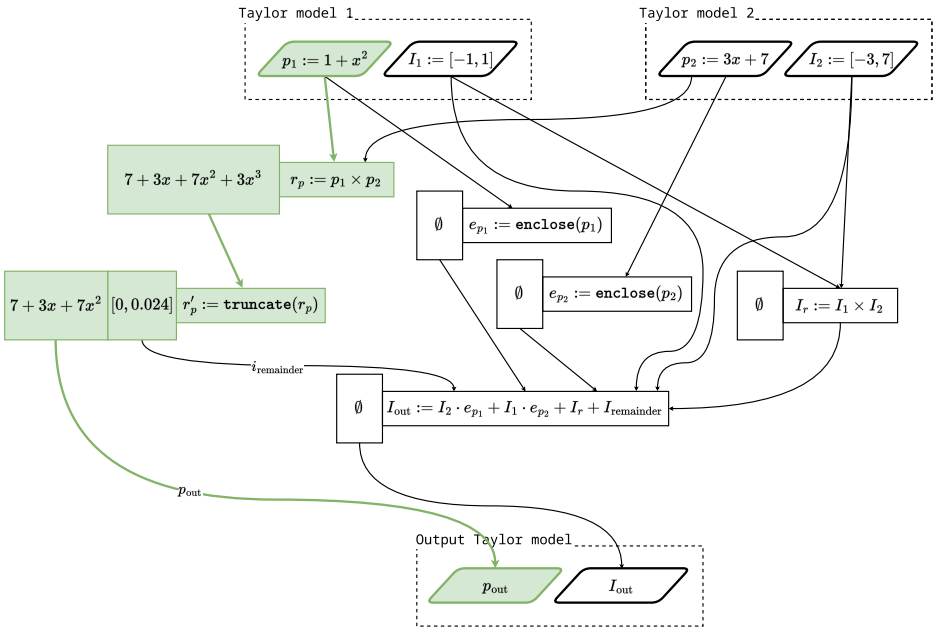


Figure 3.4: Dependency graph for Taylor model multiplication of (p_1, I_1) and (p_2, I_2), with only the polynomial component of the result requested, new computations are shown in green

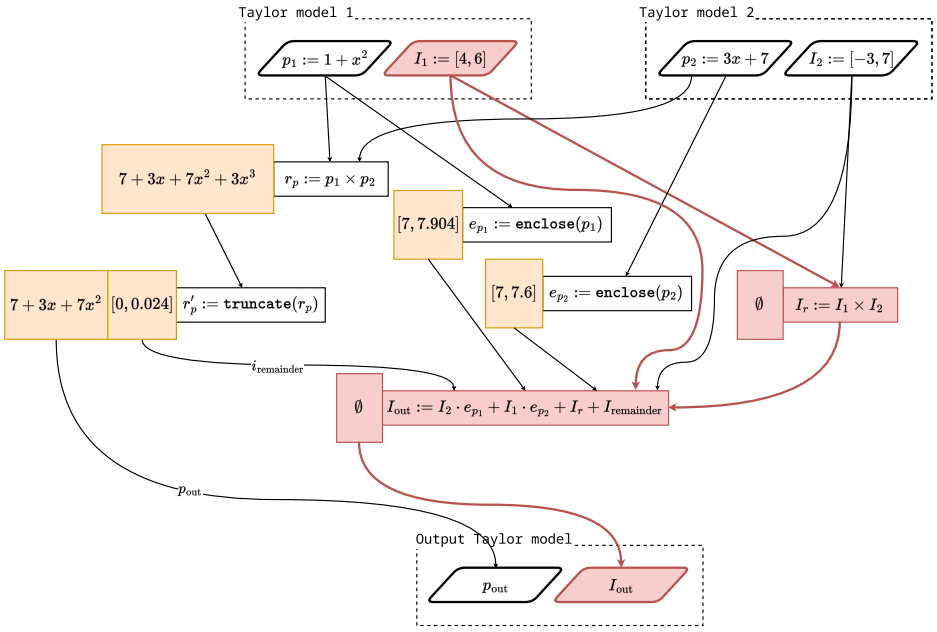


Figure 3.5: Dependency graph for Taylor model multiplication of (p_1, I_1) and (p_2, I_2), with a changed input interval I_1. Invalidated caches are shown in red, and retained caches in orange

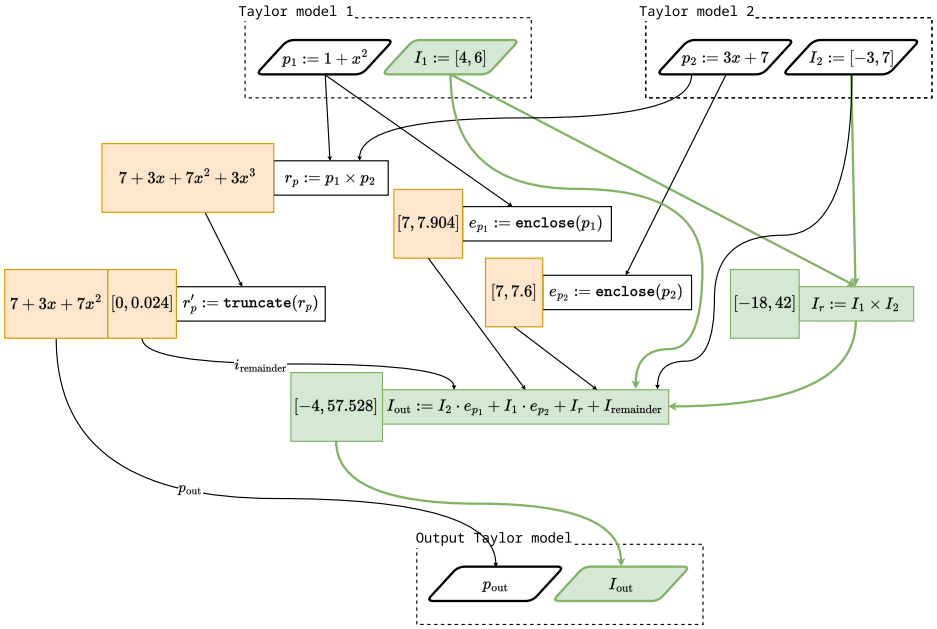


Figure 3.6: Dependency graph for Taylor model multiplication of (p_1, I_1) and (p_2, I_2) with some example values for both, with the interval I_1 changed and the result interval requested. New computations are shown in green, and retained/reused caches in orange

Now assume that we have computed both interval and polynomial components, but then we modified one of the input intervals, say I_1 without loss of generality, we can invalidate the results reliant on that value. The graph after this invalidation is shown in Figure 3.5, with invalidated vertices shown in red, and unchanged cached values shown in orange.

Then if we request the polynomial component, the cached value is used, and if the interval component is requested we re-use the values that aren't dependent on the changed interval, and only recompute those that do. This effectively results in a partially applied Taylor model multiplication operation, with the only argument being this changed value, that is then executed when the interval component is requested in Figure 3.6.

This is a commonly encountered scenario during Taylor model arithmetic, as this is the situation that occurs when performing Picard iteration to contract the remainder interval of an approximate solution to an ODE. To reiterate, the Picard operator is,

$$\mathbb{P}_F(g)(\vec{x}_0, t) = \vec{x}_0 + \int_0^t F(g(\vec{x}_0, s), s)\, ds. \tag{2.5}$$

In order to perform remainder interval contraction once we have established an order k approximation p_k, we 'guess' an initial appropriate remainder interval I_0, such that $B(p_k) \subset I_0$. We then perform Picard iteration on the Taylor model (p_k, I_0), which by the assumption that p_k is converged results in an expression of the form,

$$\begin{aligned} \mathbb{P}_F(p_k + I_i)(\vec{x}_0, t) &= \vec{x}_0 + \int_0^t F(p_k(\vec{x}_0, s) + I_i, s)\, ds \\ &= p_k + I_{i+1} \end{aligned} \qquad \text{such that } I_{i+1} \subseteq I_i \tag{3.1}$$

But in order to do the Taylor model substitution $F(p_k(\vec{x}_0, s) + I_i, s)$, many multiplication operations are required, wherein only the interval inputs to the operations change between iterations. This presents an ideal scenario for the application of partial evaluation as outlined above, as we can avoid repeating expensive operations such as polynomial multiplication when none of its inputs have changed.

3.2 Data structures

3.2.1 Polynomials

Polynomial representations fall into two primary categories, each of which is suitable for different situations and requirements: sparse and dense representations.

One consideration is the coefficient type for the polynomials. As we are interested in over-approximations we necessarily require polynomials to be able to output a range of values. However, this could be achieved via a variety of techniques, such as taking intervals as coefficients to begin with or storing an upper and lower polynomial with floating-point coefficients. For simplicity, this project chose to adopt intervals as coefficients from the outset. The use of interval coefficients results in a unified representation for single-value (via an interval of $[a, a]$) and multi-value coefficients potentially within the same polynomial. This greatly simplifies the internal arithmetic logic, and with appropriately designed arithmetic operations defined on the intervals themselves, they can be treated as if they were any other data type.

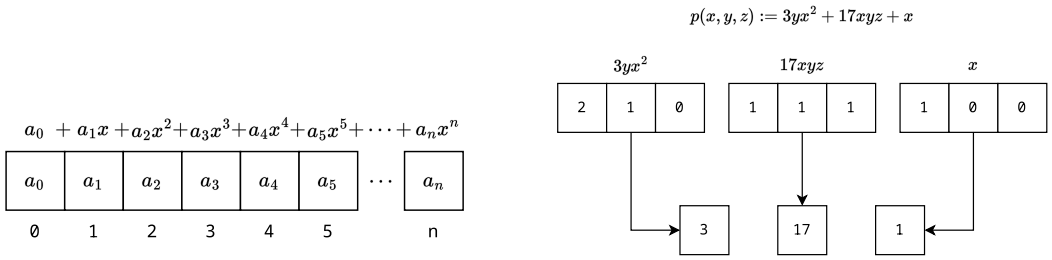
Regardless of the particular representation of polynomials, they need to be able to perform the same set of operations,

1. **Addition**: Given two polynomials p_1, p_2 of order n, m respectively, any representation needs to be able to compute their sum of order $\max\{n, m\}$, with coefficients being combined if exponents match as expected.

2. **Multiplication**: Given two polynomials p_1, p_2 of order n, m respectively, we need to be able to compute their product $p_1 \cdot p_2$ of order $n + m$.

3. **Integration**: In the algebraic sense, given a polynomial p, and bounds ℓ, u for upper and lower bounds respectively, we need to be able to compute its antiderivative at some variable x
$$\int_{\ell}^{u} p\, dx$$
As we know that p is a polynomial with real coefficients, this can be performed by simply iterating over each term in the polynomial, multiplying through by x to add a power of x, and dividing through by the power of x in that term, which should never be zero as we do not allow negative exponents in any representation.

4. **Exponentiation**: In order to allow polynomials to be substituted into one another, as is required for Picard iteration for example, we need to be able to raise a



(a) Dense polynomial representation for an arbitrary univariate polynomial of degree n, with coefficients a_i for $i \in [0,n]$

(b) Sparse polynomial representation for an example multivariate polynomial

Figure 3.7: Structural diagrams comparing dense representations and sparse representations for polynomials

polynomial to a given exponent, for example given

$$p(x,y) := 1 + x^2 + x^3 + y \tag{3.2}$$

$$q(x) := 3 + x^3 \tag{3.3}$$

To calculate $p(q(x),y)$ we need to be able to compute $q(x)^2$ and $q(x)^3$. In order to reduce the complexity of either implementation, this is most easily done by expanding the exponents into repeated multiplication, and then simply re-using the existing polynomial multiplication code.

5. **Evaluation**: Given a set of values for each variable, we should be able to compute the value of the polynomial at that point. For example, if we have a polynomial $p(x) := 1 + x^2$, we need to be able to compute $p(2) = 5$. Moreover, we need the flexibility to evaluate polynomials *at other polynomials*, for example $p(3+x) = 1 + (3+x)^2 = 10 + 6x + x^2$.

3.2.1.1 Dense representation for simple univariate polynomials

In order to represent a polynomial densely, one stores the coefficients of the polynomial in an array, in which the index of each value represents an exponent, and the value the coefficient. For univariate polynomials of lower order, this representation can be efficient, such as in Figure 3.7a.

The dense representation chosen is based around an array-like structure, in which the index of an element indicates something about the exponents of the term, and the element's content indicates the coefficient of the term. This works well in the univariate case, as only having one variable means that the size of this array scales in the number of terms. However, when one introduces multiple variables to the array, it begins to consist more and more of zeros due to the exponential growth in the number of combinations of these variables. Consequently, although fast and simple, the naïve dense implementation is only appropriate for simple univariate polynomials.

Algorithms for addition, subtraction etc. are trivial in the dense case and boil down to element-wise operations of the desired type. There are multiple options for multiplication, however as the main focus of the project was not optimisation of polynomial

operations, the one chosen is a nested loop through each of the polynomials' elements, adding the pairwise multiplication of the coefficient to the element representing the addition of the two exponents.

Evaluation of a dense polynomial in this form can either be done in the obvious way of adding to a result accumulator every coefficient multiplied by the evaluation point to the appropriate power, or via Horner's method, which we explain further in Section 4.2.2.0.3. The latter provides significant improvements in the number of additions and multiplications required, and is explored further in the implementation section.

3.2.1.2 Sparse representation for multivariate polynomials

For multivariate polynomials (and of course relatively sparse univariate polynomials such as x^{2024}), sparse representation becomes more advantageous due to the exponential blow-up in the number of exponent combinations in the number of variables. There are a number of different approaches with their own advantages and disadvantages for sparse representations, such as tree representations, sorted monomial arrays, or exponent to coefficient maps.

As the use-case in this project is to serve as an example data-type for the graph component, the priority was clarity of code and ease of implementation given the limited time-frame. Therefore, although more performance and space-efficiency could be gleaned from a more elaborate implementation, the primary use-case for this class is as a demonstrator of the optimisations. Thus, in the interests of development time and clarity of code, the architecture chosen was a mapping between exponent vectors and coefficients, i.e. a sparse polynomial with dense monomial representation.

After opting for a sorted mapped implementation, there are two main implementation decisions left to be made: the sorting schemes for the variable order inside the exponent vectors, and the order of the exponent vectors inside the map.

Variable ordering is typically a lexicographic one, and as the focus of this project is not optimising polynomial representations, this is the schema chosen. Lexicographic ordering in this case means comparison by dictionary order letter-by-letter in each variable name, such that $a < b$, and $ba < bb$, with the order lowest-to-highest.

Secondly, the ordering of the terms was inspired by the implementation proposed in Maple 14 [19], a graded lexicographic sort. This sort is done in two parts, it initially sorts based on the sum of all the exponents in each monomial, and then in order to tie-break and sort within a set total it uses a typical lexicographic sort on the exponents.

Opting for a lexicographically sorted tree map, this project attempts to balance performance with compactness. By ensuring lexicographic sorting we enforce a canonical ordering on the polynomial's monomial terms when the map is iterated over linearly, simplifying operations such as multiplication and comparison. Furthermore, storing the elements in a tree map, versus say a sorted array, look-up and existence-checking is highly efficient and is $O(1)$ in the cost of hashing the coefficient list.

3.2.1.3 Prioritisation

As should be clear from the preceding discussion, a sparse representation supporting multiple variables requires significantly more engineering that a mono-variate dense one. To this end, we initially prioritise the simper dense representation with an assumed variable of x in order to have some representation with which to test simple arithmetic on simple mono-variate Taylor models, with a sparse multivariate implementation as a stretch goal. Although it bears mention that the vast majority of systems of interest are multivariate, and it would serve as a more illustrative benchmark of the advantages of the optimisation techniques outlined to test them on multivariate systems.

3.2.2 Intervals

Although intervals are conceptually simpler objects to design in comparison to some of the others in this project, there are still some components to the design that need to be treated carefully. This is primarily in regards to the definition of their arithmetic, which can be unintuitive at first glance. Intervals need to be able to do a more limited set of operations as compared to polynomials, as outlined in Section 2.1.1.1:

1. **Addition, Subtraction**: Given two intervals, the structure should be able to automatically create a new interval to compute the sum or subtraction, this is relatively straightforward, defined as

$$[a_1, b_1] \pm [a_2, b_2] := [a_1 \pm a_2, b_1 \pm b_2] \tag{2.1.2}$$

   Whilst automatically ensuring that the appropriate floating-point rounding modes are used to ensure that intervals are always over-approximative.

2. **Multiplication**: As defined previously, interval multiplication is the minimal and maximal values attainable from a value from each of the two operand intervals

$$I_1 \cdot I_2 := \left[\min_{x \in I_1, y \in I_2} xy, \max_{w \in I_1, z \in I_2} wz \right] \tag{2.3}$$

   In the case where all interval bounds are positive, multiplication can be expressed intuitively as,

$$[a, b] \cdot [c, d] = [ac, bd].$$

   But, this straightforward calculation does not hold for intervals that include negative numbers. For example, consider the expression of $[-1, 1] \cdot [-7, 9]$, the result is *not* $[7, 9]$ as the former rule would indicate, but rather $[-9, 9]$. We can rewrite the mathematical definition in a more programmatic way as

$$[a, b] \cdot [c, d] = [\min\{ac, ad, bc, bd\}, \max\{ac, ad, bc, bd\}]. \tag{3.4}$$

3. **Exponentiation**: Exponents require additional thought, as upon first glance we can just expand the definition of an exponent and apply the definition of multiplication, i.e. that

$$[-1, 1]^2 = [-1, 1] \cdot [-1, 1] = [-1, 1].$$

However, this assumes that for $[a,b]^n$ we have n free variables each of which lies inside the interval $[a,b]$, but it actually refers to a single variable $x \in [a,b]$. So the aforementioned $[-1,1]^2$ is actually equal to $[0,1]$. This definition only diverges from the naïve expanded-multiplication approach when the interval contains zero, and even then only when the exponent is even as negative numbers to an odd power remain negative. Therefore, we can emulate this behaviour by checking for these conditions, setting the lower-boundary to zero if they are met and calculating the upper boundary as normal.

3.2.3 Taylor models

Taylor models are reliant on the implementation for polynomials and intervals, as they are of course just pairs of both. However, they still have their own design considerations to consider, and several specific requirements to fulfil in order to be considered useful or even correct.

Firstly, they need to be able to somehow group a polynomial and an interval together, and then use these values to perform Taylor model computation, as introduced in Section 2.1.1.2

1. **Multiplication**: If the Taylor model is made up of a computational vertex for its interval, polynomial, or both, then when we perform multiplication it should automatically create the whole network of vertices as seen in Figure 3.3, and return a Taylor model consisting of the two deferred result vertices seen at the bottom.
2. **Addition**: Again, generically the Taylor model structure should be able to perform addition with other Taylor models, creating the appropriate computational vertices on the graph and returning a result Taylor model without doing any eager computation, and in such a way that the user does not have to worry about managing these vertices themselves.
3. **Integration**: Taylor model integration is a multi-step process that involves the truncation and polynomial integration of the polynomial part of the Taylor model, before bounding the discarded highest term into the interval component, see Definition 2.1.7. This is an essential operation for many applications of Taylor models, and should perform all these operations automatically.

Chapter 4

Implementation

In this chapter we present implementation details, based on the previously outlined design. We outline some technical challenges and trade-offs taken, and motivate the choice of technology where appropriate.

4.1 Choice of language

We selected `C++` as the language of implementation due to its high-performance, the expressive templating system available in modern `C++` (`C++11` onwards), and the preexisting extensive ecosystem of verification tools. We outline a brief rationale for choosing `C++`, and consider some alternatives.

1. **Template Metaprogramming**: Templates are a meta-programming concept present in `C++`, and refers to the ability to define a class or function once generically with a type parameter, and then specialise it later to reduce repetitive code [1]. In the context of this project, the ability to specify one template vertex class that handles the evaluation and temporary value storage, regardless of the type it is specialised on, means that we can write concise, expressive code that is compiled down to be identical to a less expressive, less readable version.
2. **Smart Pointers**: Features such as smart pointers (e.g. `std::unique_ptr` and `std::shared_ptr`) facilitate safe but fast memory management without the user having to interact with the internals of the library in order to avoid leaks.
3. **Mature Ecosystem**: As the final product of this project is a library, it is important to select a language with a pre-existing user-base in order for the library to be useful. Implementing the library in `C++` obviously allows the user to interact with the library in `C++` as a first-class citizen, but also permits future work creating a Python library binding, if so desired.
4. **Performance**: `C++` is a highly performant language, and is highly suited to computationally intensive tasks such as Taylor model arithmetic.

Alternatives considered include Rust and Python. Rust is a high-performance systems programming language with an innovative approach to memory management [18].

An implementation in Rust would likely perform similarly to one in C++, as they are typically comparable in benchmarks [26], and the trait-based system of abstraction Rust offers would be suited for a project heavily employing polymorphism such as this. However, the vast majority of work in the field is still done in C/C++, and therefore an implementation in Rust would limit the scope of the library's applicability.

Another option was Python, a dynamically typed interpreted language [24], with a very mature ecosystem of packages and wide support. However, due to the interpreted nature of the language, the performance of algorithms in Python typically lags behind C++ by one or even two orders of magnitude [22]. Thus, although an implementation in Python may be very concise and invisible, any performance gained by employing these optimisations will be entirely outweighed by the overhead of the language itself, and it was therefore deemed unsuitable.

4.2 Data structures

4.2.1 Intervals

Implementation details for intervals follow the design closely, and there are few technical decisions to consider. We provide a simplified class declaration, explain the member variables, outline the list of required arithmetic operations, and explain their implementation details.

```
class interval : public containable {
public:
  interval() = default;
  interval(double lower, double upper) : m_lower(lower), m_upper(upper) {}
  interval(double value) : m_lower(value), m_upper(value) {}
private:
  double m_lower, m_upper;
};
```

Listing 4.1: Simplified class declaration for the `interval` class

1. **Addition, Subtraction**: Given two intervals `a` and `b`, addition is performed by calculating `a.m_lower + b.m_lower` and `a.m_upper + b.m_upper`, where the private member variables are accessible as all arithmetic operators are defined as `friend` methods. These bounds are then used to construct a new interval.
2. **Multiplication**: Given two intervals `a` and `b`, we calculate all four products of the possible permutations, i.e. lower times lower, lower times upper, etc. and take the minimum of them for the lower bound, and the maximum for the upper bound.
3. **Exponentiation**: For an interval `a` and an exponent `n`, we check if `n` is even and if the interval `a` contains 0, and if so we set the lower boundary to zero, otherwise we calculate the exponent of the lower bound. We then calculate the exponent of the upper bound, and then return an interval with these two bounds.

The only factor that needed special attention was the rounding direction to maintain the over-approximative nature of Taylor model arithmetic, that is, 'downwards' for the lower bound, and 'upwards' for the upper bound.

This can be accomplished using the C++ standard library, specifically the <cfenv> header functions `std::fegetround` and `std::fesetround` [28] functions.

We implemented this by attaching these methods to set and restore the floating-point rounding mode to the singleton `tf::context` class, which can then be called around interval floating-point operations that require this specific rounding.

For example, to ensure interval multiplication captures the boundary values properly for the upper and lower bounds, we change the rounding mode to `FE_UPWARD` when calculating the upper bound, and to `FE_DOWNWARD` when calculating the lower bound.

4.2.2 Dense univariate polynomials

We chose to target dense, univariate polynomials before attempting the sparse, multivariate case in order to ensure we implemented a functional 'minimum viable product' (MVP). This approach allowed us to quickly have a prototype executing simple polynomial and later Taylor model operations, ensuring that we had a concrete conceptual understanding before spending development time implementing more difficult sections of functionality.

4.2.2.0.1 Addition Addition for two dense polynomials represented by arrays a and b of degree n and m respectively is a process of first right-padding the smaller of the two arrays with zeros to be the same size as the larger one, and then performing element-wise addition to combine the two arrays into a single coefficient array.

4.2.2.0.2 Multiplication Multiplication is performed via a simple implementation of the Standard Algorithm/Schoolbook for Multiplication. The coefficients of the two polynomials are multiplied inside a nested loop, with the results added to the appropriate coefficient of some accumulator result polynomial.

Definition 4.2.1 (Schoolbook algorithm for polynomial multiplication)**.** Given polynomials $P(x) = \sum_{i=0}^{n} a_i x^i$ and $Q(x) = \sum_{j=0}^{m} b_j x^j$, their product $R(x) = P(x) \cdot Q(x)$ is calculated as:

$$R(x) = \sum_{k=0}^{n+m} \left(\sum_{i+j=k} a_i b_j \right) x^k$$

This algorithm has a complexity of $O(nm)$, which although sub-optimal is acceptable given the typical sizes of polynomials in the domain of interest.

4.2.2.0.3 Evaluation We consider two methods for evaluation of a dense (univariate) polynomial, the typical method and Horner's method. The typical method simply involves iterating over the monomial terms of the polynomial, calculating the value to the power of the term of the monomial, multiplying by the coefficient, and adding to an accumulator coefficient type, which in this case is an interval. Although this is how humans evaluate polynomials, and is the most obvious algorithm, it requires n additions

and $\frac{n(n+1)}{2}$ multiplications, in comparison to Horner's method which only requires n of both operations.

Horner's method for polynomial evaluation works by factoring each term of the polynomial, such that each 'level' of brackets contains a constant term and an x multiplied by the remainder of the bracketing. In practice this transformation is in the form

$$a_0 + a_1x + a_2x^2 + \cdots + a_nx^n = a_0 + x\Big(a_1 + x\Big(a_2 + \cdots + x(a_{n-1} + xa_n)\cdots\Big)\Big).$$

This then allows for the evaluation of the polynomial via an accumulator initialised to the coefficient of the highest degree and a single `for` loop over all coefficients but the highest of the array in reverse order. It proceeds by multiplying the accumulator by the evaluation point x and adding the next coefficient, the final value of the accumulator is then the value of the polynomial at that point.

4.2.2.0.4 Integration We implemented indefinite integration via a simple right shift of the elements in the polynomial, before dividing each coefficient by its index, excluding the new zero-th element. We then provide a helper function for definite integration by first performing indefinite integration, and then returning the difference of the antiderivative evaluated at the two bounds.

4.2.2.0.5 Exponentiation As we only support natural number exponents, we implemented exponentiation by expanding the exponent out into repeated multiplication. Although the ability to compute fractional or negative exponents would be useful for specification of more complex ODEs, we felt that the additional development time required to implement such functionality was better spent elsewhere due to the scope of the project being limited to simpler ODE systems.

4.2.3 Sparse Polynomials

4.2.3.0.1 Addition In the context of sparse polynomials, addition is performed by merging the two sequences of monomials from each operand, either combining coefficients if two terms match, or adding a new monomial term if they do not.

4.2.3.0.2 Multiplication Multiplication of sparse polynomials is starkly different from the dense case. This is primarily due to the fact that the size of the result grows quadratically [23]. Therefore, as the theoretical complexity is only $O(t^2 \log D)$ [23], where t is the total number of terms of both polynomials and D is the maximum degree, we opt for the classical or schoolbook algorithm for multiplication. This is similar to the dense case, involving repeated multiplications by each term and addition to an accumulator, but with more complicated logic to assign each result coefficient to the correct position. The use of a hash-map facilitates fast lookup and insertion in this use-case, and ensures that we do not accumulate multiple terms with identical exponents.

4.2.3.0.3 Integration We implemented indefinite integration by looping over all monomials, adding one to the power of the variable we are integrating with respect to, and then dividing the coefficient by this new power. Definite integration is done by first calculating the indefinite integral, and then computing the difference between the upper and lower bound of the interval the user is integrating over. By reusing the evaluation code in this way, we support both a numerical bound, e.g. $\int_0^5$, but variable bounds, e.g. $\int_0^t$.

4.2.3.0.4 Exponentiation Similar to the algorithm for dense polynomials, we implemented exponentiation by expanding the exponent out into repeated multiplication. This approach has the same limitations as those laid out in Paragraph 4.2.2.0.5.

4.2.3.0.5 Evaluation Evaluating sparse polynomials, assuming that we maintain the same global variable set, is a simple iterative process over each monomial. Assuming we are evaluating a polynomial at some point $x = x_0$, at each monomial the appropriate power of x_0 is calculated based on the power of x in the monomial, or skipping if it is zero, multiplying the existing coefficient by this value, before finally setting the power of x to zero. This process is optimised by introducing a power-cache, a look-up table that stores the values of x_0^i as they are calculated, and uses the cached result if it has been previously calculated. This is another scenario in which memoisation grants a large increase in performance, as if we have a polynomial such as

$$3x^2y + 7x^2y^2 + 14x^2y^3$$

Then we need only calculate x_0^2 once to evaluate the polynomial at $x = x_0$, rather than three times.

4.2.3.1 Variable Spaces

Ideally the multivariate class would be able to take in any polynomial and represent it without any additional input from the user. This could be achieved by having each object track its own variable-space, i.e. the mapping between indices of exponent vectors and the variables they represent ($x \mapsto 0, y \mapsto 1$ etc.), and then having these two spaces combined and pruned when doing operations involving multiple polynomials. In practice, however, this proved to be difficult to implement, and therefore the decision was made to require the variable space be defined before any polynomials were, in effect freezing the representations in place. The means that unfortunately, the user has to call `tf::multivariate::add_variable` function on either a single variable string or a vector of such, before they can create a polynomial using those variables. This has the further disadvantage that if one creates a polynomial, and then adds variables to the space, the first polynomial is effectively corrupted. However, for the use-case of the class, processing models with known variables ahead of time, this was an acceptable trade-off, but is one we would revisit if given more development time.

4.2.3.2 Data Structure

The internal representation is a map-based design, mapping exponent combinations to coefficients. `C++`'s standard library provides both sorted and unsorted maps in the

form of `std::map` and `std::unordered_map`, and although there are better performing open-source components available such as the popular parallel-hashmap project [21], the standard library implementation was the simplest to use and provided adequate performance, given that coefficient look-up was not a bottleneck in performance.

4.2.3.3 Lexicographic Ordering

The algorithms outlined above for operations on sparse polynomials rely on a consistent ordering of variables inside the exponent vectors. This is due to the fact that the exponent vectors are used as keys in the coefficient hash-map, and as different permutations of the same vector hash to different values, if the ordering between polynomials was inconsistent then duplicate values could accumulate. The most common ordering for variables is lexicographic ordering, that is ordering based on pairwise comparison by position in the alphabet, e.g. `"a" < "b"`, with further characters only being compared to tie-break, e.g. `"aaa" < "aab" < "aac"`.

4.2.3.4 Parsing

In order to make the input of multivariate polynomials as easy as possible, we implemented a parser using the parsing and lexing library Lexy [13]. Lexy offers a combination of the expressiveness that one gets with high-level libraries, with the performance of lower-level approaches. This is achieved by doing as much processing at compile-time as possible, as every component is a `constexpr`. The format assumed for the polynomials matches the one specified by Flow* [12], with monomials in the form `([exponents]*)*\*([variable^exponent]*)*`, i.e. `3*x*y - 2*x^2*z`.

Although an initial implementation was attempted using regular expressions, it made for a cleaner and more readable implementation to construct actual parser machinery with labels and productions, rather than attempting to catch all edge-cases inside a large, magic, unreadable regular expression.

4.3 Dependency graph management

In order to use the DG to implement partial and lazy evaluation, we must first construct and maintain the relationships, whilst ensuring appropriate lifetime and memory locations for all the involved objects.

4.3.1 Vertices

Vertices contain a type parameter `T` that indicates the underlying type they represent, and a standard-library `optional<T>` field to serve as either the container for the value of a concrete vertex, or as the value cache for a computational vertex. It exposes a function `v` that checks whether a cached value is present, returning it if so, and if one is not present, as should only be the case if the vertex is a computational vertex, it calls the expression associated with it that is represented by a `std::functional` object. It then takes the value returned by this function, caches it, and returns it. This system allows

for the vertex class to represent any underlying type, as itself has no knowledge of the item it contains, increasing the re-usability of the code-base to further applications.

```
template <typename T> class vertex {
public:
  vertex(T &&value) : m_value(std::move(value)) {}
  vertex(function<T()> &&calc_func) : m_calc_func(std::move(calc_func)) {}
  bool has_value() const;
  const T &v();
  void invalidate();
  void replace(T &&new_value);

  operator T() const { return v(); }
  const T operator()() { return v(); }

private:
  optional<T> m_value;
  optional<function<T()>> m_calc_func;
};
```

Listing 4.2: Simplified class definition of `vertex<T>`

In order to create a `vertex<T>` object, one must either supply in the constructor a concrete value of the type `T`, or a lambda expression with a return type matching `T`.

However, in order to allow the user to swap-out the value of a vertex, or to replace a relationship in a computation, another layer of abstraction outside the ones required in the design-phase was required. By replace a relationship, say I have three vertices containing $x = 1$, $x = 2$, $y = 3$, then I have another that contains the expression $x + y$, then the user should be able to swap out whether that x points to $x = 1$ or $x = 2$ without having to create entirely new vertices.

One piece of functionality considered was to employ a C++ template programming pattern called the 'Curiously recurring template pattern' [1], in which a class inherits from its template parameter. In this context, this would mean the use of `std::static_assert` to ensure that the type parameter `T` was an appropriate data-type, or more specifically a virtual class of `interval` or `polynomial`. This would allow the speculative and base versions of the types to be treated identically, as they would both implement the expected operators and functions defined on the appropriate data-type's virtual class. However, this approach produced opaque and hard-to-read code, and required too much type-specific code in order to refer to the base class given the appropriate interface. Therefore although conceptually cleaner than the implementation decided-upon, it was dropped. If more time was allotted, this would be something that would further reduce the amount of user-input to almost zero, and make the library completely transparent.

```
template <typename T> class vertex<T> : public T {
    static_assert(std::is_base_of<polynomial, T>::value
    ↪  std::is_base_of<interval, T>::value), "T must be be a containable
    ↪  type!");
    // rest of implementation...
};
```

4.3.2 Graph

In order to maintain the edges of the DG, we need to maintain the edge relationships between the vertices. Although there are many ways to do this, due to the relatively low average connectivity of the graph, the relationships are stored via a simple adjacency list system. This system is indexed via a unique ID assigned to each vertex as it is added to the graph structure, and moved into the graph's `vector` of vertices. The relationships are mapped using a standard library `std::unordered_map` object, unordered to increase the insert performance, as the relations are rarely iterated over linearly and as such do not need to maintain a consistent ordering.

The system uses modern `C++` `std::move` semantics in order to ensure that the graph structure 'owns' all the objects passed to it, rather than them being inside the scope from which they are added. This avoids any potential memory management issues by ensuring that all objects have a lifetime that matches that of all possible computations done on them, and as a result the user doesn't have to do memory management on their own objects, increasing the usability of the library overall.

Furthermore, the class contains a helper method to convert the structure to Mermaid format [29], a language to describe graphs with a rendering engine to render them on the fly. This allows for the easy visualisation of the dependencies and computations as the user creates them, allowing for a better understanding of what is going on under-the-hood if desired.

4.3.2.1 Alternative graph structures considered

We also considered an alternative configuration for the dependency graph with separate vertices for each operation, in the style of a binary expression tree. Although conceptually simpler to traverse than the one based on computational vertices, this proved: difficult to implement due to issues storing different data-types in the same structure; to be significantly less expressive than the one based on lambda expressions; and involved a significant amount of boilerplate code to implement. As a result, although requiring fewer modern `C++` features to implement, and a marginally less complex, more brute-force implementation, we opted for the more complex and fully featured implementation outlined above.

4.3.3 Wrappers

As the graph and its vertices lie on the heap, the user could interface directly with the vertices themselves via their pointers, and have each vertex itself hold its identifier and a reference to the graph to which it belongs. However, due to the semantics of `C++`, one cannot overload arithmetic operators such as `operator+` where both operands are pointers; at least one of the operands must be a reference or a variable. Therefore, as we want the user to be able to interact with results of vertices in exactly the same fashion as their underlying types, another layer of abstraction was required, that we call a wrapper.

The `wrapper<T>` class is relatively simple, containing an ID of the vertex it represents, and a pointer to the graph containing that vertex. This class can then be assigned to,

moved, used, deleted, etc. all as if it were the type `T` that underlies it via overloaded operators, and it will automatically interface with the graph to ensure that the structure is maintained. This class lives on the stack in the user's own code, which given its small size of a graph pointer and identifier should result in little performance degradation versus keeping larger structures such as polynomials on the stack.

```
template <typename T> class wrapper {
public:
  wrapper(graph *g, size_t id) : m_graph(g), m_id(id) {}
  wrapper(const wrapper &other) = default;
  T operator()() const { return m_graph->get<T>(m_id); }
  vertex<T> *operator[]() const { return m_graph->get_vertex<T>(m_id); }
  vertex<T> *vertex() const { return m_graph->get_vertex<T>(m_id); }
  explicit operator size_t() const { return m_id; }

  void operator=(T &&new_value);
  void operator=(size_t new_id);
  void operator=(wrapper<T> &&other) noexcept;
  void operator=(wrapper<T> &other);
};
```

Listing 4.3: Simplified class definition of `wrapper<T>`

The overloaded assignment operators call the `.update` function on the graph with the supplied replacement value, with different semantics for left and right references, new values, and new IDs.

4.3.3.0.1 Common arithmetic operators

In order to facilitate the use of the library, we define a number of arithmetic operators on wrappers of polynomials, intervals, and Taylor models. These functions then allow the user to express arithmetic in exactly the same way as the underlying type, whilst maintaining the advantages of the DG structure. For example, the following

```
auto p1 = N(multivariate("1 + x^2"), g);
auto p2 = N(multivariate("y^3"), g);
auto p3 = p1 * p2;
```

Is functionally identical to the following, more verbose code:

```
auto p1 = N(multivariate("1 + x^2"), g);
auto p2 = N(multivariate("y^3"), g);
auto p3 = L(p1() * p2(), g, p1, p2);
```

We employ the `=` capture-by-copy capture semantic when constructing lambda expressions for deferred vertices. This allows the lambda's contents access to the variables of the containing scope via copying, rather than by reference as with the `&` capture. Although this has a performance impact versus passing by reference, the values being copied are the small `tf::wrapper` structures, containing only the pointer to a graph and the vertex ID, rather than the entire vertex and potentially its large cached result. This was done to ensure that the values used inside the expressions represent the vertices as they are when it is defined, rather than changing later on if the wrapper(s) referenced are replaced with another value, avoiding a confusing user experience. For example, if capture-by-reference semantics were used, then a loop-based implementation of Picard

iteration which assigns to a variable p_k on each iteration, would result in a recursive dependency of p_x on itself. This would violate the acyclic nature of the graph, causing undefined, incorrect behaviour.

4.3.4 Taylor models

Taylor models are implemented as a class containing wrappers, pointing to vertices representing its polynomial and interval components.

```
class taylormodel {
public:
  wrapper<multivariate> p;
  wrapper<interval> I;
};
```

We then defined a series of constructors, avoiding requiring the user to have to interact with the underlying graph directly if desired, these accept any combination of

- A concrete polynomial or interval or both as a `rvalue` reference, using `std::move` semantics to move the supplied concrete values to concrete vertices on the supplied graph `g`.
- Preexisting wrappers of either polynomial or interval, i.e. existing results that we wish to pair together as a Taylor model.

Note that it can construct valid Taylor models from both just an interval or a polynomial, allowing the use of Taylor model arithmetic on these more primitive types directly. This lets the user perform all computation on the Taylor model level, rather than requiring them to directly interact with the underlying structures, both programatically and mathematically, unless they wish to do so.

We then defined C++ operators for multiplication, addition, subtraction, and negation, which automatically construct the appropriate vertices and relationships on the DG required to perform such operations in an efficient way, such as that laid out in Figure 3.3. In addition to a cast operator to turn a Taylor model into a polynomial if desired,

```
wrapper<multivariate> taylormodel::operator()() {
  return L(p() + I(), p.get_graph(), p, I);
}
```

4.4 Testing

We employed functional testing throughout the project, seeking to validate our results with those of examples provided in Chen, Abraham and Sankaranarayanan (2012) [9]. This approach allowed us to be confident in the correctness of each successively more complex version of the library, as we started with the simplest case (univariate polynomials, etc.) and progressively expanded functionality to include more complex systems. Behaviour was tested via the Catch2 [7] testing framework. This allowed us to do traditional test-driven development for the data representation components of the project via unit-testing, in addition to making claims on the number of calculations done by a snippet via a custom `REQUIRE_CALCULATIONS` macro.

Chapter 5

Evaluation

In this chapter we evaluate the capabilities of the library versus the specification, and provide some benchmarks to demonstrate its efficacy.

5.1 Project goals revisited

5.1.1 Optimisations

5.1.1.1 Lazy evaluation

Lazy evaluation was achieved as outlined in Section 3.1.1 by use of the dependencies tracked via the DG. To demonstrate this, we took utilised our testing macro to ensure that the number of calculations performed agreed with the number expected. For example, the following test passes, with the zero argument to `REQUIRE_CALCULATIONS` indicating that no computation is being done when the unused result w is created.

```
1  auto p = N(multivariate("1 + x^2"), g);
2  auto q = N(multivariate("1 + y^2"), g);
3  REQUIRE_CALCULATIONS(0, auto w = p * q);
```

5.1.1.2 Memoisation

Memoisation is achieved by always returning the value of the `std::optional` field of the vertex if present, and the library therefore avoids repeated calculation if none of the inputs have changed, and therefore no invalidations have been propagated, as in Figure 3.5.

```
1  auto p = N(multivariate("1 + x^2"), g);
2  auto q = N(multivariate("1 + y^2"), g);
3  auto w = p * q;
4  REQUIRE_CALCULATIONS(5, w().print());
5  REQUIRE_CALCULATIONS(0, w().print());
```

5.1.1.3 Partial evaluation

Partial evaluation also works as expected, with only the required components being re-calculated upon some inputs being changed. For example, the following toy test-cases utilising the `REQUIRE_CALCULATIONS` testing-macro demonstrate the reduction in computations required when a limited subset of the inputs to a computation are changed, versus without requiring the entire expression to re-evaluated.

```
    // assume set-up as required
  auto p = N(multivariate("1 + x^2"), g);
  auto q = N(multivariate("1 + y^2"), g);
  auto i = N(interval(-1, 1), g);
  auto r = p + q;
  r = r + i;
  REQUIRE_CALCULATIONS(2, r().print(););
  i = interval(-17, 70);
  REQUIRE_CALCULATIONS(1, r().print(););
```

In contrast, without utilising the DG structure, any modifications in any inputs forces a full-recalculation, as partial results cannot be salvaged, as shown below:

```
  auto p = multivariate("1 + x^2");
  auto q = multivariate("1 + y^2");
  auto i = interval(-1, 1);
  multivariate r;
  REQUIRE_CALCULATIONS(2, r = p + q; r = r + i; r.print(););
  REQUIRE_CALCULATIONS(2, i = interval(-17, 70); r = p + q; r = r + i;
                       r.print(););
```

To demonstrate the tangible performance improvement in terms of computation time, we can take advantage of Catch2's `BENCHMARK` macro in a slightly more complex polynomial operation scenario (See Appendix B.2 for the full listing). This demonstrates that although additional overhead is introduced to manage the dependencies, even on the smallest examples there is a net benefit in terms of total computation time. The results, presented in Table 5.1, demonstrate a 73% improvement in performance with partial evaluation enabled versus full recalculation.

Table 5.1: Benchmark results for with and without partial evaluation for polynomial addition, all values to 3 s.f.

Benchmark Name	**Mean (μs)**	**Std Dev (μs)**
Partial Re-Evaluation	19.6	2.06
Full Re-Evaluation	33.9	3.33

This advantage becomes more pronounced in the context of two-stage Picard iteration for the approximation of ODEs. Here, the focus was on the optimisation of the computation of the remainder interval of the computed Taylor model. This was tested on the 1-dimensional ODE $\dot{x} = 1 + x^2$, with the parameters $k = 10, x(0) \in [0, 1/5]$, and a single time-step of $t \in [0, 1/10]$. Benchmarking revealed a substantial speedup, with partial evaluation outperforming the typical case by a factor of 12, as shown in 5.2. This confirms the hypothesis laid out in the specification, that the application of these

techniques to Taylor model-specific arithmetic yields significant performance improvements of approximately 10x. The complete code listing of this test-case can be found in Appendix B.1.

Table 5.2: Benchmark results for with and without partial evaluation for Picard iteration, all values to 3 s.f., see Appendix B.1 for the code listing

Benchmark Name	**Mean (ms)**	**Std Dev (μs)**
Typical Picard Iteration	45.2	676.0
Partial Picard Iteration	3.71	130.0

If we separate out the lazy and partial evaluation optimisations, we observe the expected speed-ups from their separate employment. We performed Taylor model multiplication on two example Taylor models, and compared the time taken to obtain just the interval or polynomial component, and an equivalent implementation done with the base classes and no optimisations at all. These results are presented in Table 5.4. As expected only requesting the polynomial component was significantly faster than computing the interval component, which matches expectations, as we avoid computing a significant proportion of the graph from Figure 3.3. Examining purely partial evaluation, we

Table 5.3: Benchmark results for various TM multiplication methods, all values to 3 s.f., see Appendix B.5 for the code listing

Benchmark Name	**Mean (μs)**	**Std Dev (μs)**
Lazy TM multiplication, only polynomial requested	331.0	69.0
Lazy TM multiplication, only interval requested	804.0	85.7
Eager TM multiplication	1320.0	25.8

employ the `volatile` keyword to ensure we force an initial computation of both the interval and polynomial components. We then randomly assign a new interval to one of the operand Taylor models, and then request either the polynomial or interval component of the result again. These results are presented in Table 5.4. Requesting the polynomial component involves no recomputation, and is therefore almost instant, and requesting the interval component is significantly faster than doing the entire computation again.

Table 5.4: Benchmark results for various TM multiplication methods with interval changes, all values to 3 s.f., see Appendix B.4 for the code listing

Benchmark Name	**Mean (μs)**	**Std Dev (ns)**
Partial TM multiplication, interval changed, polynomial only	6.56	852.0
Partial TM multiplication, interval changed, interval only	120.0	3320.0
Eager TM multiplication	1310.0	7910.0

These results are, in our view, enough to justify the utility of the targeted optimisations on Taylor models, as the performance improvements from their employment are non-trivial, and represent an area for significant performance gains in an application employing a large number of Taylor model computations.

5.1.2 User experience

Improving the user experience when using this class of optimisations was a key design goal. We attempted to balance aggressive optimisation with producing a non-intrusive intuitive interface. Although some progress has been made here, there is definite scope for improvement.

5.1.2.1 Invisibility of operations

The use of optimisations via the use of a suite of macros, whilst not entirely invisible, are designed to be less obtrusive than the alternatives available. We have implemented a system where the majority of the complexity of dependency management is abstracted away from the user. This allows for cleaner user-code, where the user can focus on the mathematical operations without worrying about the underlying implementation details.

However, although less burdensome than alternatives, the interface is far from perfect. It still demands a level of awareness of the user to provide a list of dependencies of their expressions, and having every expression wrapped in a macro detracts from the readability of the code, see Section 5.1.1 or the Appendices for examples.

Moreover, this method introduces an additional layer of abstraction, and whilst all efforts have been made to ensure that these objects act identically to their underlying data-types, this is not perfect. Future work on this library to address this could include exploring implementation of a Domain Specific Language (DSL) for specification of computations in order to alleviate the requirement for dependency specification, or a plug-in for the popular `C++` compiler Clang to inspect each lambda and extract dependencies at compile-time.

5.1.3 Data representations

5.1.3.1 Intervals

We support intervals specified by either two floating-point values for lower and upper bounds, or by supplying a string and utilising the parser. The user can then perform all required interval arithmetic (as specified in Section 2.1.1.1) such as multiplication, in addition to more complex operations such as checking if one interval is contained within another. Furthermore, the interval class properly sets the floating-point rounding mode in order to ensure the over-approximative guarantee of Taylor model arithmetic.

5.1.3.2 Polynomials

Our library effectively handles univariate polynomials and associated arithmetic operations. We evaluate the implementations for dense and sparse representations separately.

5.1.3.2.1 Dense univariate polynomials Univariate polynomials can be constructed by providing an array-like structure of coefficients in a little endian style, i.e. that lower indices are lower order terms. It then allows the user to perform all the expected arithmetic operations on them specified in Section 3.2.1, such as multiplication, evaluation, and integration.

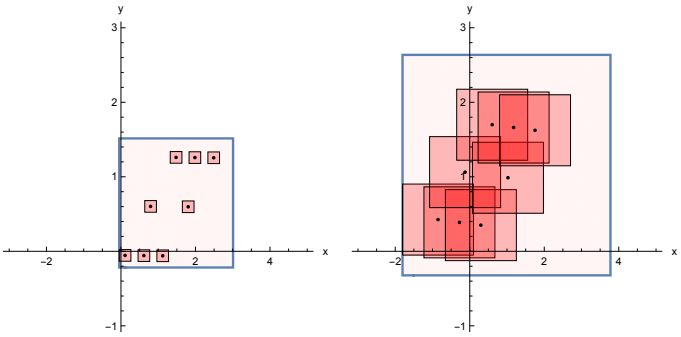


Figure 5.1: Over-approximation enclosure boxes for the whole Lotka-Volterra system at $t = 0.6$ and $t = 0.7$, with some boundary points mapped from the input space of $x \in [0,1], y \in [0,2]$ to their interval boxes in the output space. See Appendix B.3 for the code listing, and Appendix A.1 for more time steps

5.1.3.2.2 Sparse multivariate polynomials The user can construct sparse polynomials iteratively by providing pairs of exponent vectors and coefficients, or they can utilise the string parser to specify polynomials as strings. They can then perform all the expected arithmetic operations laid out in Section 3.2.1, including indefinite and definite integration, evaluation in any number of variables, and multiplication.

5.1.3.3 Taylor models

The user can specify Taylor models as an interval, a multivariate polynomial, or a pair of both. Where the single-value constructors create the appropriate 'identity', e.g. supplying just a polynomial results in an interval component of $[0,0]$. We only support specifying multivariate polynomials in Taylor models, as the class superseded the functionality of the dense polynomial one.

The user can then perform the expected Taylor model arithmetic operations, as specified in Section 2.1.1.2, and the class will automatically ensure that the DG structure is maintained in such a way to ensure partial and lazy evaluation without the user's involvement whatsoever.

5.1.4 Application to Lotka-Volterra

In order to demonstrate the functionality of the library for Taylor model arithmetic, we implemented a variety of single time-steps for the Lotka-Volterra system of ODEs that model a predator-prey relation. We then created a Taylor model approximation for the system via two-step Picard iteration, and used Mathematica [15] to produce Figure 5.1 from the results, including the mappings of boundary points at the various time-steps. Although the library is capable of single-step propagation like this, additional techniques such as those outlined in Section 6.3.0.0.1 are required in order to do effective multi-step propagation without the system's parameter intervals 'blowing up' to the point where they make no useful claims about the behaviour of the system. Case in point, note how much larger the boxes for the $t = 0.7$ step are vs the $t = 0.6$ step, their larger size indicating we can make less precise statements about their behaviour.

Chapter 6

Conclusion

6.1 Final remarks

This project has implemented a Taylor model arithmetic library in C++ with automatic lazy and partial evaluation, with a significantly improved user interface. We achieve this through the implementation of a dependency graph that is automatically managed without the user's involvement, leveraging modern C++ features to reduce complexity, increase expressiveness, and increase performance.

We have implemented a sufficient subset of Taylor model arithmetic to make the library useful for a large proportion of common Taylor model arithmetic scenarios. Including classes for univariate and multivariate polynomials in dense and sparse representations respectively, and a class to handle interval arithmetic. We implemented a parser for intervals and multivariate polynomials to increase productivity and reduce the labour involved in specifying a Taylor model problem.

We hope that this library will serve as a useful tool for both students and researchers working with Taylor models. By streamlining the process of optimised Taylor model arithmetic, we aim to increase productivity by reducing processing times and increasing expressiveness, allowing users to focus on the mathematics of their problems rather than on the optimisation of their code.

In this chapter we present some limitations of the library, and propose possible future work that could improve the functionality and usability of the library.

6.2 Limitations

6.2.1 Optimisation transparency

Although the interface provided to the user is significantly less cumbersome than the one demanded by comparative libraries such as Flow*, it still demands some user input in order to function. For instance, the user must specify a generic function's dependencies in order to have them be appropriately linked in the graph structure. This process is made less cumbersome with the addition of macros, but to take advantage of the ability

of the deferred computation vertices to execute any arbitrary piece of code, the user must still specify its dependencies at that point in time, for example:

```
auto result = L(x().integrate("x"), g, x);
```

This is an area that could be avoided by making the computational vertices more application specific, and thus restricting their content by disallowing the user from constructing their own. However, we believe that the small task of specifying dependencies is a small price to pay for the level of flexibility offered.

6.2.2 Automatic vertex-reuse

Ideally, if a deferred computation vertex was requested to be created that already matched the exact computation of another vertex on the graph, with the same dependency arguments, then rather than creating a new but identical vertex in the graph we would return a wrapper pointing to the existing vertex. This would reduce the size of the graph and facilitate additional computational savings by ensuring maximum reuse of previously calculated results. However, the expressions inside deferred computational vertices are constructed via C++ lambda functions, which are assigned a unique type and cannot be inspected at compile-time. This means that the ideal system, i.e. one of direct inspection of the lambdas, to determine whether or not we can re-use an existing vertex, does not work. As a workaround, optionally each deferred vertex can take a label that is generated based by a supplied function descriptor and the identifiers of its dependencies/arguments, which can then be searched for inside the graph, but unfortunately this requires more user-interaction than desired.

Therefore, as a compromise the library implements arithmetic operators for wrappers on common data types in a file `common_operators.{hpp,cpp}`, that automatically fill out a label and create the appropriate vertices on the graph of the operands. For example, if one has two intervals and computes their product,

```
auto I1 = N(interval(-1,1), g);
auto I2 = N(interval(-5,0), g);
auto I3 = I1 * I2;
```

Then if later we create another wrapper for the product of I_1 and I_2, the original computational vertex would be re-used:

```
auto I4 = I1 * I2;
assert(I3.id() == I4.id());
```

However, generic lambda expressions are always assumed to be distinct and new vertices created regardless. For example, the following will be result in two distinct vertices being created:

```
// assume x, y are defined
auto a = L(x().integrate("t", "0", y()), g, x, y);
auto b = L(x().integrate("t", "0", y()), g, x, y);
REQUIRE(a.id() != b.id());
```

This could potentially be alleviated by inspection of the intermediary LLVM Intermediary Representation via a Clang plugin. Such a plugin would be able to inspect

the IR representation of each requested lambda, and assign an identifier based on the computations it performs rather than anything manually supplied. This would allow vertex re-use to be significantly broader and net decreased graph sizes and increased performance, unfortunately this lay outside the scope of the project given the time allocated, but is something to be explored in future work.

6.3 Future work

We consider some potential enhancements to the library that would improve its utility and performance.

6.3.0.0.1 Coordinate transformations In order to increase the rate of convergence, especially when doing multiple steps in time, tools such as Flow* use a technique of coordinate transformations. As we typically consider Taylor models centred around the origin (see Definition 2.1.1), we obtain significantly better convergence results when we keep the domains 'centered' at the origin. This involves transforming the n-box at each time-step to map to the unit interval $[-1, 1]$ per Taylor model via a process called *preconditioning* [8, pp. 67–71]. We then track each of these transformations in a single separate Taylor model, effectively resulting in two Taylor models where we would otherwise have one.

Unfortunately, this proved to be outside the scope of this project, and as such the library lacks the ability to converge as quickly as Flow*, see Section 5.1.4 for an example. This would be an extremely useful addition to the project, as it would allow the library to compete on the same models and output as Flow*.

6.3.0.0.2 Parallelism The same dependency information captured by the DG that we utilise the obtain partial and lazy evaluation could be used to implement parallelism. As we know which results rely on what, we could compute results that are not reliant on each other in parallel, avoiding data-races that are common when parallelising interdependent computations such as these. For example, we could compute the polynomial component of a Taylor model multiplication in parallel with the interval components, the two interval bounds of the polynomials and the interval product in parallel, only computing the truncation error introduced after we have computed the polynomial portion. This is an area that would again yield significant speed-ups, but lay outside the scope of the project.

6.3.0.0.3 Hybrid systems Hybrid systems are those with both continuous and discrete behaviour, and are often called 'Cyber-physical systems' [8]. For example, a model of water flow through a channel could have the continuous behaviour of the rate of water flow, and the discrete behaviour of whether or not a gate is shut constricting the flow. Analysis of these systems presents significantly more of a challenge versus their continuous counterparts, with which this paper was primarily focused. A potential extension to the library would be the capability to handle such hybrid models, in effect creating a like-for-like competitor to Flow*, utilising our approach to Taylor model arithmetic.

Bibliography

[1] David Abrahams and Aleksey Gurtovoy. *C++ template metaprogramming: concepts, tools, and techniques from Boost and beyond*. The C++ in-depth series. 5. print: Addison-Wesley, 2009. 373 pp. ISBN: 978-0-321-22725-6.

[2] Luis Benet et al. *JuliaIntervals/TaylorModels.jl: v0.6.2*. Version v0.6.2. 14th Apr. 2023. DOI: 10.5281/ZENODO.2613102. URL (visited on 18/10/2023).

[3] Martin Berz. 'From Taylor series to Taylor models'. In: Beam stability and nonlinear dynamics. Santa Barbara,California (USA), 1997, pp. 1–23. DOI: 10.1063/1.53493. URL (visited on 18/10/2023).

[4] Martin Berz. 'Modern Map Methods in Particle Beam Physics'. In: ().

[5] Martin Berz and Kyoko Makino. 'Verified Integration of ODEs and Flows Using Differential Algebraic Methods on High-Order Taylor Models'. In: *Reliable Computing* 4.4 (1st Nov. 1998), pp. 361–369. ISSN: 1573-1340. DOI: 10.1023/A:1024467732637. URL (visited on 18/10/2023).

[6] Jeff Bezanson et al. *Julia: A Fast Dynamic Language for Technical Computing*. 23rd Sept. 2012. DOI: 10.48550/arXiv.1209.5145. arXiv: 1209.5145[cs]. URL (visited on 18/10/2023).

[7] *catchorg/Catch2*. original-date: 2010-11-08T18:22:56Z. 7th Mar. 2024. URL (visited on 07/03/2024).

[8] Xin Chen. 'Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models'. PhD thesis. RWTH Aachen, Mar. 2015.

[9] Xin Chen, Erika Abraham and Sriram Sankaranarayanan. 'Taylor Model Flowpipe Construction for Non-linear Hybrid Systems'. In: *2012 IEEE 33rd Real-Time Systems Symposium*. 2012 IEEE 33rd Real-Time Systems Symposium (RTSS). San Juan, PR, USA: IEEE, Dec. 2012, pp. 183–192. ISBN: 978-1-4673-3098-5. DOI: 10.1109/RTSS.2012.70. URL (visited on 18/06/2023).

[10] Earl A. Coddington and Norman Levinson. *Theory of Ordinary Differential Equations*. Google-Books-ID: LvNQAAAAMAAJ. McGraw-Hill, 1955. 456 pp. ISBN: 978-0-07-099256-6.

[11] Thomas H. Cormen, ed. *Introduction to algorithms*. 3rd ed. OCLC: ocn311310321. Cambridge, Mass: MIT Press, 2009. 1292 pp. ISBN: 978-0-262-03384-8 978-0-262-53305-8.

[12] *Flow*: A Verification Tool for Cyber-Physical Systems*. Flow*: A Verification Tool for Cyber-Physical Systems. URL (visited on 06/03/2024).

[13] *foonathan/lexy: C++ parsing DSL*. URL (visited on 06/03/2024).

[14] Yoshihiko Futamura. 'Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler'. In: *Higher-Order and Symbolic Computation* 12.4

(1st Dec. 1999), pp. 381–391. ISSN: 1573-0557. DOI: 10.1023/A:1010095604496. URL (visited on 18/10/2023).

[15] Wolfram Research Inc. *Mathematica, Version 14.0*. URL.

[16] Neil D. Jones. 'An introduction to partial evaluation'. In: *ACM Computing Surveys* 28.3 (Sept. 1996), pp. 480–503. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/243439.243447. URL (visited on 23/03/2024).

[17] Kyoko Makino and Martin Berz. 'TAYLOR MODELS AND OTHER VALIDATED FUNCTIONAL INCLUSION METHODS'. In: *International Journal of Pure and Applied Mathematics* 4 (Jan. 2003), pp. 379–456.

[18] Nicholas D Matsakis and Felix S Klock II. 'The rust language'. In: *ACM SIGAda ada letters*. Vol. 34. Number: 3. ACM, 2014, pp. 103–104.

[19] Michael Monagan and Roman Pearce. 'Sparse polynomial multiplication and division in Maple 14'. In: *ACM Communications in Computer Algebra* 44.3 (28th Jan. 2011), pp. 205–209. ISSN: 1932-2240. DOI: 10.1145/1940475.1940521. URL (visited on 24/01/2024).

[20] Nedialko S. Nedialkov and John D. Pryce. 'Solving Differential-Algebraic Equations by Taylor Series (I): Computing Taylor Coefficients'. In: *BIT Numerical Mathematics* 45.3 (Sept. 2005), pp. 561–591. ISSN: 0006-3835, 1572-9125. DOI: 10.1007/s10543-005-0019-y. URL (visited on 01/04/2024).

[21] Gregory Popovitch. *greg7mdp/parallel-hashmap*. original-date: 2019-03-02T13:55:44Z. 10th Mar. 2024. URL (visited on 11/03/2024).

[22] *Python 3 vs C++ g++ - Which programs are fastest?* URL (visited on 10/03/2024).

[23] Daniel S. Roche. 'What Can (and Can't) we Do with Sparse Polynomials?' In: *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*. 11th July 2018, pp. 25–30. DOI: 10.1145/3208976.3209027. arXiv: 1807.08289[cs]. URL (visited on 08/03/2024).

[24] Guido van Rossum and Fred L. Drake. *The Python language reference*. Release 3.0.1 [Repr.] Python documentation manual / Guido van Rossum; Fred L. Drake [ed.] Pt. 2. Hampton, NH: Python Software Foundation, 2010. 109 pp. ISBN: 978-1-4414-1269-0.

[25] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. Google-Books-ID: gL34DwAAQBAJ. MIT Press, 20th Feb. 2004. 931 pp. ISBN: 978-0-262-22069-9.

[26] *Rust vs C++ g++ - Which programs are fastest?* URL (visited on 11/03/2024).

[27] *Stack Overflow Developer Survey 2023*. Stack Overflow. URL (visited on 19/10/2023).

[28] *std::fegetround, std::fesetround - cppreference.com*. URL (visited on 07/03/2024).

[29] Knut Sveidqvist and Contributors to Mermaid. *Mermaid: Generate diagrams from markdown-like text*. original-date: 2014-11-01T23:52:32Z. Dec. 2014. URL (visited on 07/03/2024).

Appendix A

Additional figures

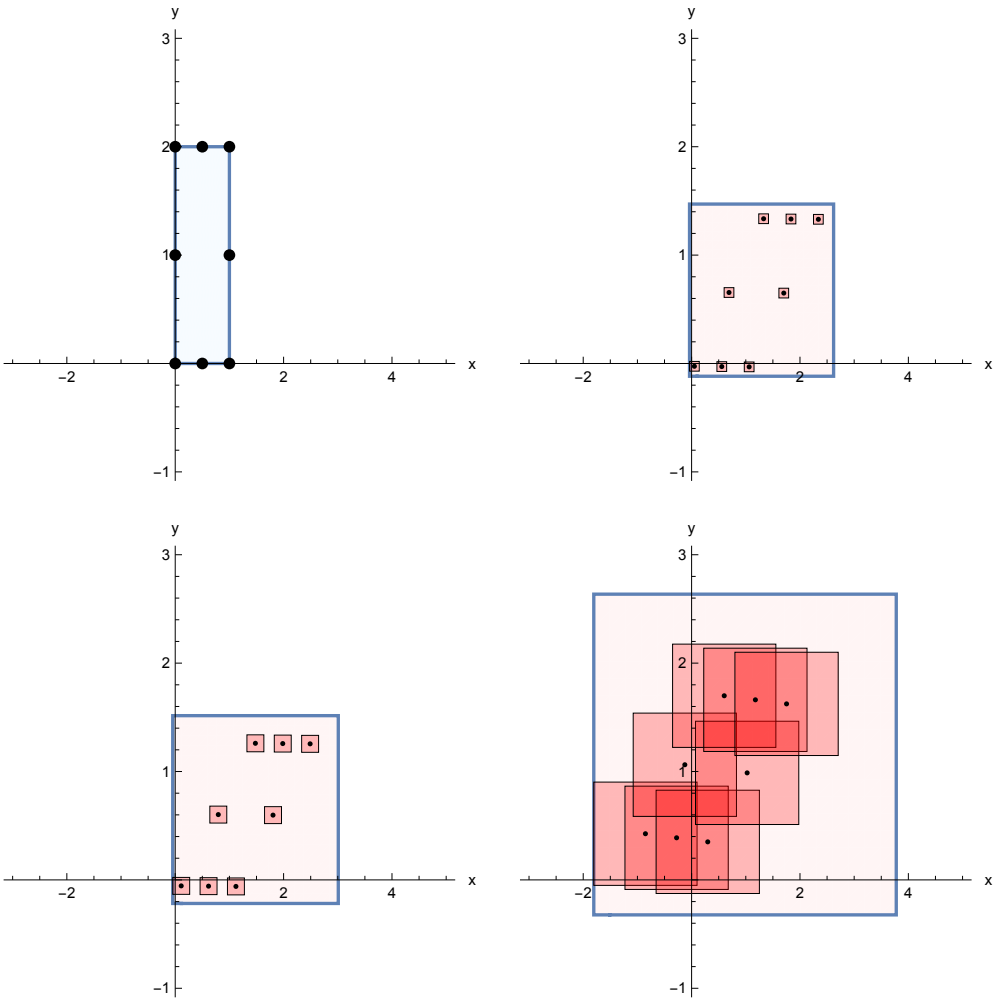


Figure A.1: Initial interval box and over-approximation boxes for the Lotka-Volterra equations at $t = 0.4, 0.5, 0.6, 0.7$, with some boundary points mapped from the input space mapped to the output space. Elaborated on in Section 5.1.4

Appendix B

Code listings

Listing B.1: Benchmarking code used to produce the results in Table 5.2

```
#include "util.hpp"
#include <catch2/benchmark/catch_benchmark_all.hpp>
#include <catch2/catch_test_macros.hpp>
#include <taylorflow.hpp>

using namespace tf;
using namespace std;

TEST_CASE("Partial evaluation of picard iteration", "[partial-picard]") {
  auto g = context::get_instance().get_graph();
  multivariate::add_variable(vector<string>{"x", "s", "t", "I0"});
  const int k = 10;
  const double tol = 1e-10;
  {

    auto ode = multivariate("1 + x^2");
    auto x0 = interval(0, 1. / 5);
    auto g_i = multivariate(x0);
    auto t = interval(0, 1. / 10);
    BENCHMARK("Typical picard-iteration") {
      for (int i = 0; i < k; i++) {
        g_i = (ode.evaluate("x", g_i).integrate("s", "0", "t"))
                  .discard_high_exponents(k) +
              x0;
      }
      auto i0 = interval(-1, 1);
      auto i0_next = i0;
      do {
        i0 = i0_next;

        auto new_terms =
            ode.evaluate("x", g_i + "I0").integrate("s", "0", "t") - g_i;
        i0_next = new_terms
                      .evaluate(map<string, interval>{
                          {"x", x0}, {"t", t}, {"I0", i0}})
                      .constant_term();
      } while (max(abs(i0.lower() - i0_next.lower()),
```

```
                    abs(i0.upper() - i0_next.upper())) > tol);
      return make_pair(g_i, i0);
    };
  }
  {
    auto ode = N(multivariate("1 + x^2"), g);
    auto x0 = N(interval(0, 1. / 5), g);
    auto g_i = L(multivariate(x0()), g, x0);
    const auto t = interval(0, 1. / 10);

    BENCHMARK("Partial picard iteration") {

      for (int i = 0; i < k; i++) {
        g_i = L(ode().evaluate("x", g_i()), g, g_i);
        g_i = L(g_i().integrate("s", "0", t), g, g_i);
        g_i = L((g_i() + x0()).discard_high_exponents(k), g, g_i, x0);
      }
      auto i0 = N(interval(-1, 1), g);
      auto i0_next = i0;

      do {
        i0 = i0_next;
        auto new_terms = L(
            ode().evaluate("x", g_i() + "I0").integrate("s", "0", "t") -
            ↪  g_i(),
            g, g_i, ode);
        i0_next = L(new_terms()
                        .evaluate(map<string, interval>{
                            {"x", x0()}, {"t", t}, {"I0", i0()}})
                        .constant_term(),
                    g, new_terms, x0, i0);
      } while (max(abs(i0().lower() - i0_next().lower()),
                   abs(i0().upper() - i0_next().upper())) > tol);
      return make_pair(g_i(), i0());
    };
  }
}
```

Listing B.2: Benchmarking code used to produce the results in Table 5.2

```
TEST_CASE("Full vs. partial re-evaluation benchmark", "[partial-eval]") {
  multivariate::add_variable(vector<string>{"x", "y"});
  SETUP_GRAPH(g)

  {
    auto p = N(multivariate("1 + x^2"), g);
    auto q = N(multivariate("1 + y^2"), g);
    auto i = N(interval(-1, 1), g);
    auto r = L(p() + q(), g, p, q);
    r = L(r() + i(), g, r, i);
    BENCHMARK("Partial re-evaluation") {
      i = interval(-1, 1);
      r();
      i = interval(-17, 70);
      r();
    };
  }
```

```
  {
    auto p = multivariate("1 + x^2");
    auto q = multivariate("1 + y^2");
    auto i = interval(-1, 1);
    multivariate r;
    BENCHMARK("Full re-evaluation") {
      r = p + q;
      r = r + i;

      i = interval(-17, 70);
      r = p + q;
      r = r + i;
    };
  }
}
```

Listing B.3: Example code used to compute a single time-step of the Lotka-Volterra equations, see Figure 5.1 and surrounding discussion

```
#include "context.hpp"
#include "interval.hpp"
#include "multivariate.hpp"
#include "taylormodel.hpp"
#include <catch2/catch_test_macros.hpp>
#include <map>
#include <taylorflow.hpp>

using namespace tf;
using namespace std;

TEST_CASE("Lokka-volterra equations", "[lotka-volterra]") {
  multivariate::add_variable(vector<string>{"x", "y", "s", "t"});
  context::get_instance().add_tm_domain("x", interval(0, 1));
  context::get_instance().add_tm_domain("y", interval(0, 2));
  context::get_instance().add_tm_domain("s", interval(0.2, 0.2));
  context::get_instance().add_tm_domain("t", interval(0.2, 0.2));

  const int k = 5;

  auto g_unique = graph::new_unique();
  auto g = g_unique.get();

  auto x = N(interval(0, 1), g);
  auto y = N(interval(0, 2), g);
  auto x_tm = taylormodel(x, g);
  auto y_tm = taylormodel(y, g);

  taylormodel gx(multivariate(x()), g);
  taylormodel gy(multivariate(y()), g);

  for (int i = 0; i < k; i++) {
    auto gx_subst = 2 * gy - gx * gy;
    auto gy_subst = 0.5 * gx * gy - gy;
    auto gx_int = x_tm + gx_subst.integrate("s", "0", "t");
    auto gy_int = y_tm + gy_subst.integrate("s", "0", "t");
    auto gx_lowered = gx_int.lower_to_order(k);
```

```
    auto gy_lowered = gy_int.lower_to_order(k);

    gx = gx_lowered;
    gy = gy_lowered;
  }
  gx.I = interval(-5, 5);
  gy.I = interval(-5, 5);
  auto n_calcs = context::get_instance().get_calculations();
  for (int i = 0; i < 10; i++) {
    auto gx_subst = 2 * gy - gx * gy;
    auto gy_subst = 0.5 * gx * gy - gy;
    auto gx_int = x_tm + gx_subst.integrate("s", "0", "t");
    auto gy_int = y_tm + gy_subst.integrate("s", "0", "t");
    auto gx_lowered = gx_int.lower_to_order(k);
    auto gy_lowered = gy_int.lower_to_order(k);
    gx = gx_lowered;
    gy = gy_lowered;
  }
}
```

Listing B.4: Partial Taylor model multiplication benchmarking code, used to produce the results in Table 5.4

```
#include <catch2/benchmark/catch_benchmark_all.hpp>
#include <catch2/catch_test_macros.hpp>
#include <catch2/generators/catch_generators_all.hpp>
#include <string>
#include <taylorflow.hpp>

using namespace tf;
using namespace std;

TEST_CASE("Partial-only TM multiplication benchmark",
          "[partial-only-benchmark]") {
  multivariate::add_variable(vector<string>{"x", "y"});
  context::get_instance().add_tm_domain("x", interval(0, 1));
  context::get_instance().add_tm_domain("y", interval(0, 1));

  auto g = context::get_instance().get_graph();

  const int k = 5;
  const string x_str = "1 + x^2 + 3*x^4 + -7*x^5 + 1475*x^6*y^2 + 3*x^7*y^3";
  const string y_str = "71 + -3*y^2 + 2*y^3 + 5*y^4 + -7*y^5 + 11*y^6 +
   ↪  13*y^7";

  auto x = taylormodel(multivariate(x_str), interval(-1, 1), g);
  auto y = taylormodel(multivariate(y_str), interval(-1. / 2, 1. / 2), g);
  auto z = x * y;
  volatile auto z_p = z.p(); // force initial eval
  volatile auto z_i = z.I(); // force initial eval
  BENCHMARK("Partial TM multiplication, interval changed, polynomial only") {
    // generate a random new interval for x
    auto l = GENERATE(take(1, random(-10, 10)));
    auto u = GENERATE(take(1, random(-10, 10)));
    x.I = interval(l, u);
    return z.p();
```

```
  };
  BENCHMARK("Partial TM multiplication, interval changed, interval only") {
    // generate a random new interval for x
    auto l = GENERATE(take(1, random(-10, 10)));
    auto u = GENERATE(take(1, random(-10, 10)));
    x.I = interval(l, u);
    return z.I();
  };

  auto x_p = multivariate(x_str);
  auto y_p = multivariate(y_str);
  auto x_i = interval(-1, 1);
  auto y_i = interval(-1. / 2, 1. / 2);
  BENCHMARK("Eager TM multiplication") {
    auto l = GENERATE(take(1, random(-10, 10)));
    auto u = GENERATE(take(1, random(-10, 10)));
    x_i = interval(l, u);

    auto z_p_non_trunc = x_p * y_p;
    auto [z_p, z_p_rem] = z_p_non_trunc.split(k);
    auto z_i =
        x_i * y_i +
        x_p.evaluate(context::get_instance().get_tm_domain()).constant_term()
        ↪ *
            y_i +
        y_p.evaluate(context::get_instance().get_tm_domain()).constant_term()
        ↪ *
            x_i +
        z_p_rem.evaluate(context::get_instance().get_tm_domain())
            .constant_term();
    return make_pair(z_p, z_i);
  };
}
```

Listing B.5: Lazy Taylor model multiplication benchmarking code, used to produce the results in Table 5.3

```
#include <catch2/benchmark/catch_benchmark_all.hpp>
#include <catch2/catch_test_macros.hpp>
#include <string>
#include <taylorflow.hpp>

using namespace tf;
using namespace std;

TEST_CASE("Lazy-only TM multiplication benchmark", "[lazy-only-benchmark]") {
  multivariate::add_variable(vector<string>{"x", "y"});
  context::get_instance().add_tm_domain("x", interval(0, 1));
  context::get_instance().add_tm_domain("y", interval(0, 1));

  auto g = context::get_instance().get_graph();

  const int k = 5;
  const string x_str = "1 + x^2 + 3*x^4 + -7*x^5 + 1475*x^6*y^2 + 3*x^7*y^3";
  const string y_str = "71 + -3*y^2 + 2*y^3 + 5*y^4 + -7*y^5 + 11*y^6 +
  ↪  13*y^7";
```

```
19
20    auto x = taylormodel(multivariate(x_str), interval(-1, 1), g);
21    auto y = taylormodel(multivariate(y_str), interval(-1. / 2, 1. / 2), g);
22    BENCHMARK("Lazy TM multiplication, polynomial only") {
23      auto z = x * y;
24      return z.p();
25    };
26    BENCHMARK("Lazy TM multiplication, interval only") {
27      auto z = x * y;
28      return z.I();
29    };
30
31    auto x_p = multivariate(x_str);
32    auto y_p = multivariate(y_str);
33    auto x_i = interval(-1, 1);
34    auto y_i = interval(-1. / 2, 1. / 2);
35    BENCHMARK("Eager TM multiplication") {
36      auto z_p_non_trunc = x_p * y_p;
37      auto [z_p, z_p_rem] = z_p_non_trunc.split(k);
38      auto z_i =
39          x_i * y_i +
40          x_p.evaluate(context::get_instance().get_tm_domain()).constant_term()
            ↪ *
41              y_i +
42          y_p.evaluate(context::get_instance().get_tm_domain()).constant_term()
            ↪ *
43              x_i +
44          z_p_rem.evaluate(context::get_instance().get_tm_domain())
45              .constant_term();
46      return make_pair(z_p, z_i);
47    };
48  }
```

Appendix C

UML Diagram

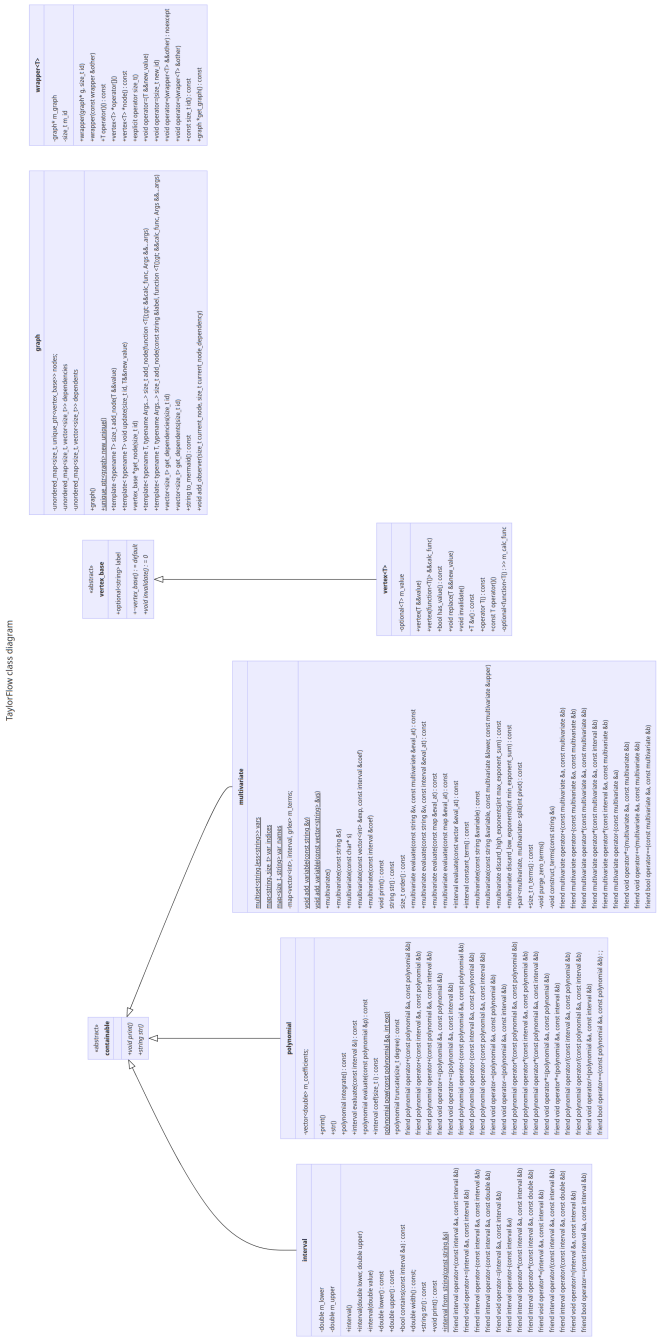


Figure C.1: UML diagram for TaylorFlow, listing all main classes, their relations, and their methods. Underlines indicate static items, pluses public items, minuses private ones, and italics virtual ones. Arrows indicate inheritance.