SOLVING HIGHER-ORDER ODES IN DRASIL

SOLVING HIGHER-ORDER ODES IN DRASIL

BY

DONG CHEN, M.Eng.

A Report

submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in partial fulfilment of the requirements

for the degree of

Master of Engineering

Master of Engineering (2022) McMaster University

(Department of Computing and Software) Hamilton, Ontario, Canada

TITLE:	Solving Higher-order ODEs in Drasil
AUTHOR:	Dong Chen M.Eng. in (Systems Engineering), Boston University, Massachusetts, USA
SUPERVISOR:	Dr. Spencer Smith and Dr. Jacques Carette
NUMBER OF PAGES:	xv, 77

Abstract

Drasil is a framework that generates software, including code, documentation, software requirement specification, user manual, and axillary files. Recently, the Drasil team has been interested in expanding its knowledge to solve higher-order ODEs. In this research, for single higher-order linear ODEs, the Drasil framework can solve them without manually extracted information. For higher-order nonlinear ODEs, the Drasil framework can solve them with manually extracted information.

Firstly, we design a flexible and reusable structure to store ODE information based on conventional mathematical knowledge. This makes it possible to reuse ODE information for documentation and for code generation. Secondly, we provide a commonality analysis of four external ODE solver libraries. The analysis includes how they solve the ODE, what algorithms they use, and what options they provide for different types of output. Thirdly, we enable the Drasil Code Generator to solve nonlinear higher-order ODEs with some manually extracted information. We created a new case study, Double Pendulum, that has a system of higher-order ODE. Further, we solve the Double Pendulum example numerically via external libraries. Lastly, for single higher-order ODEs, the Drasil Code Generator can generate code without manually extracted information.

This research accomplishes three main goals. Firstly, we capture the knowledge of

linear ODE in a flexible and reusable structure. Secondly, we expand the Drasil capability to solve higher-order ODEs with/without manually written equations. Solving single higher-order linear ODEs does not require manually extracted information. To solve nonlinear ODEs, manually extracting information from the original ODE is still required. The last one is removing the duplicated information caused by the implementation of solving ODEs.

Acknowledgements

I would like to express my deepest appreciation to my supervisors, Dr. Spencer Smith and Dr. Jacques Carette. With their guideline, I could break a complex puzzle into smaller pieces and accomplish them one by one. I spent 90% of my time in remote learning, but the quality of knowledge I received beyond learning in-person. My supervisors quickly adapted the work rhythm during the pandemic, and I greatly benefited from it.

Special thanks to my friend and colleague Jason Balaci for answering my questions and providing excellent suggestions for learning.

I am also thankful to my parents for supporting me in pursuing my second master's degree. Although my parents received very little education, they supported me in pursuing higher education mentally and financially.

Lastly, I want to thank Sophie for motivating me out of the bottom rock of my life.

Contents

Abstract **iii**

Acknowledgements **v**

Notation, Definitions, and Abbreviations **xiii**

0 Software Automation **1**

1 Introduction **3**

2 ODE Data Representation **6**

2.1 Explicit Equation 8

2.2 Matrix Form Structure 10

2.3 Input Language 15

2.4 Two Constructors 18

2.5 Display Matrix 21

3 External libraries **24**

3.1 Numerical Solutions 26

3.2 Algorithm Options 29

3.3 Output an ODE 29

4 Connect Model to Libraries **35**

4.1 Higher Order to First Order 36

4.2 Connect Drasil with External Libraries 39

4.2.1 Enable Drasil for Solving Higher-Order ODEs 39

4.2.2 Double Pendulum 44

4.3 Generate ODEInfo Automatically 47

5 Summary of Future Work **57**

5.1 Automate the Generating Process for A System of Higher-order ODEs 57

5.2 Generate an ODE Solver Object that Returns Solution on Demand . 58

5.3 Display ODEs in Various Forms 58

5.4 Allow Adaptive Steps 59

5.5 Solve ODEs as a BVP 59

5.6 Handle Dependency 60

5.7 Define ODE Solver in Drasil 60

6 Conclusion **61**

A My Appendix **63**

A.1 Constructors of DifferentialModel 64

A.2 Numerical Solution Implementation 65

A.3 Algorithm in External Libraries 68

A.4 Generated Code for Double Pendulum 71

List of Figures

2.1 NoPCM Demonstration 8

2.2 Options of Displaying an ODE 22

4.1 Double Pendulum Demonstration 44

5.1 Options of Displaying an ODE 58

List of Tables

2.1 Type Use in DifferentialModel 13

3.1 Algorithms Support in External Libraries 30
3.2 Specification for Calculations Module Returns a Finite Sequence . . . 32
3.3 Specification for Calculations Module Return an Infinite Sequence . . 34
3.4 Specification for Calculations Return a Funtion 34

4.1 Variables in Double Pendulum Example 44

A.1 Algorithm Options in Scipy - Python (15) 68
A.2 Algorithm Options in OSLO - C# (13) 68
A.3 Algorithm Options in Apache Commons Maths - Java (8) 69
A.4 Algorithm Options in ODEINT - C++ (11) 70

List of Code

2.1 NoPCM Equation for SRS 9

2.2 NoPCM Equation for the Drasil Code Generator 10

2.3 Internal Data Representation for Equation 2.2.2 14

2.4 Input Language for Equation 2.2.2 17

2.5 Explicitly Set Values for Equation 2.2.2 in DifferentialModel 19

2.6 Emulate Unknown 20

2.7 Create a Coefficient Matrix 21

3.1 A Sample Input File for Double Pendulum 31

3.2 A Output File for Double Pendulum 31

3.3 Source Code of Solving PDController in OSLO 33

4.1 Source Code for Initial Values in Drasil 40

4.2 Source Code for Initial Values in Python 41

4.3 Source Code for Initial Values in C# 41

4.4 Source Code for Returning Dimension in Java 42

4.5 Source Code for Returning a Fixed Value 43

4.6 Pseudocode for Encoding Double Pendulum's Equation 46

4.7 Source Code for Creating Identity Matrix (Highlighted in Orange) . . 49

4.8 Source Code for Creating Constant Matrix 50
4.9 Source Code for Isolating the Highest Order 52
4.10 Source Code for Canceling the Coefficient from the Highest Order . . 52
4.11 Source Code for Generating Equation 4.1.6 53
4.12 Source Code for IVP Infomation 54
4.13 Source Code for Generating ODEInfo 56

A.1 Using Input Language for the Example 2.2.2 in DifferentialModel . . 64
A.2 Source Code of Solving PDController in Scipy 65
A.3 A Linear System of First-order Representation in ACM 66
A.4 A Linear System of First-order Representation in ODEINT 67
A.5 Generate Python Code for Double Pendulum 71
A.6 Generate C# Code for Double Pendulum 72
A.7 Generate Java Code for Double Pendulum 73
A.8 Generate C++ Code for Double Pendulum 74

List of Equations

2.1.1 NoPCM Equation 9
2.2.1 Matrix Form 10
2.2.2 PDContoller Equation 11
2.2.3 PDContoller Equation in Matrix Form 12
2.2.4 Linear Higher-Order ODE 12
2.2.5 Linear Higher-Order ODE in Matrix Form 12
3.1.1 System of First-Order ODEs for PDContoller 26
4.1.6 System of First-Order ODEs in a General Form 38
4.2.1 Double Pendulum Equation 45
4.2.2 Double Pendulum Equation in a System of First-Order ODEs 45
4.3.4 Expansion of System of First-Order ODEs in Matrix Form 51

Notation, Definitions, and Abbreviations

Notation

$\mathbb{R}$	any real number in (-∞, ∞).
$\mathbb{R}^n$	a sequence that contains real numbers, n depends on the number of input values.
$\mathbb{R}^m$	a finite sequence that contains real numbers, m depends on the start time, end time, and time step.
$\mathbb{R}^\infty$	an infinite sequence that contains real numbers, ∞ is a natural number.
$\mathbb{R} \to \mathbb{R}^k$	a function takes an independent variable and outputs dependent variables.

Definitions

Drasil Framework	references the whole Drasil Project.
Drasil Code Generator	compiles captured knowledge to code.
Drasil Printer	displays captured knowledge in SRS.

Abbreviations

ACM	Apache Commons Maths
BDF	Differentiation Formula Method
BVP	Boundary Value Problem
DblPendulum	Double Pendulum
GOOL	Generic Object-Oriented Language
IVP	Initial Value Problem
NoPCM	Solar Water Heating System without PCM
ODE	Ordinary Differential Equation
PCM	Phase Change Material
PDController	Proportional Derivative Controller
RK	Runge-Kutta
SCS	Scientific Computing Software

SRS Software Requirements Specification

Chapter 0

Software Automation

From the Industrial Revolution (1760-1840) to the mass production of automobiles that we have today, human beings have never lacked innovations to improve the process. In the Industrial Revolution, we start to use machines to replace human labour. Today, we have been building assembly lines and robots in the automobile industry to reach a scale of massive production. Hardware automation has been relatively successful in the past one hundred years, and they have been producing mass products for people at a relatively low cost. With the success story of automating hardware, could software be the next target for automation? Nowadays, software is used every day in our daily life. Most software still requires a human being to write it. Programmers usually write software in a specific language and produce other byproducts, like documentation and test cases, during development. Whether in an enterprise or research institution, manually creating software is prone to errors and is not as efficient as a code generator. In the long term, a stable code generator usually beats programmers in performance. They will eventually bring the cost down because of the labour cost reduction. Perhaps this is why human beings consistently seek to

automate work.

With fairly well-understood knowledge of software, creating a comprehensive system to produce software is not impossible. Can you imagine that programmers no longer programming in the future world? In the future world, code generators will generate software. There will be a role called "code alchemist" who is responsible for writing the recipe for the code generator. The recipe will indicate what kind of software people want. In other words, the recipe is also a software requirement document that the code generator can understand. The recipe can exist in the form of a high-end programming language. Once the code generator receives the recipe, it will automatically produce software artifacts, including code, requirements, documentation and test cases. The code generator exists in the form of a compiler. The "generate everything" could be revolutionary. The Drasil framework (9) provides a proof of concept and initial steps in the direction of a generate all things approach.

Chapter 1

Introduction

Drasil is a framework that generates software, including code, documentation, software requirement specification, user manual, axillary files, and so on. We call those artifacts "software artifacts". At the moment, the Drasil framework targets generating software for scientific problems. Drasil is developed in the Haskell environment. Recently, the Drasil team has been interested in expanding its knowledge to solve a higher-order ordinary differential equation (ODE). It would be difficult to directly add ODE knowledge into the Drasil framework because this requires Drasil to have codified knowledge for ODE, which Drasil currently does not have. Thus, we believe a compromised way to solve a higher-order ODE is to generate a program that solves the ODE numerically using external libraries. There are three main reasons why we want to do that:

1. Scientists and researchers frequently use ODEs as a research model in scientific problems because they come up so often in describing natural phenomena. Building a research model in software is relatively common, and the software that the Drasil

framework generates can solve scientific problems. Thus, expanding the Drasil framework's potential to solve all ODEs would solve many scientific problems. Currently, Drasil can only solve first-order ODEs numerically.

2. Many external libraries are hard to write and embody much knowledge, so the Drasil team wants to reuse them instead of reproducing them.

3. Another reason is that the Drasil team is interested in how the Drasil framework interacts with external libraries. Once the team understands how to interact between the Drasil framework and external libraries, they will start to add more external libraries. In this way, we will unlock the potential for the Drasil framework to solve more scientific problems.

The Drasil framework neither captures ODE knowledge nor solves higher-order ordinary differential equations. Previous researcher (3) incorporated solving a first-order ODE, but it only covers a small area of the knowledge of ordinary differential equations. Most first-order ODE knowledge was captured by following the knowledge-based approach (6). Adding higher-order linear ODEs into the Drasil framework will expand the area where it has never reached before. Therefore, my research will incorporate higher-order linear ODEs in a complex knowledge-based and generative environment that can link to externally provided libraries.

To solve a higher-order linear ODE, we have to represent ODEs in the Drasil database. On the one hand, users can input an ODE as naturally as writing an ODE in mathematical expressions. On the other hand, they can display the ODE in the style of conventional mathematical expressions. The data representation will preserve the relationship between each element in the equation. Then, we will analyze the commonality and variability of selected four external libraries. This analysis

will lead us to know how external libraries solve ODEs, what their capabilities are, what options they have, and what interfaces look like. Then, we need to bridge the gap between the Drasil ODE data representation and external libraries. The Drasil ODE data representation cannot directly communicate with external libraries. Each library has its own interface in terms of solving ODEs. The existing gap requires a transformation from the Drasil ODE data representation to a generic data form before solving ODEs in each programming language. Finally, users can run software artifacts to get the numerical solution of the ODE.

Before conducting my research, the Drasil framework could solve explicit equations and numerically solve a first-order ODE. After my research, the Drasil framework will have full capability to solve a higher-order linear ODE numerically. Cases study of NoPCM and PDController will utilize a newly created model to generate programs to solve a higher-order linear ODE in four different programming languages. In addition, we will explore the possibility of solving a system of ODE numerically. We will introduce a new case study, Double Pendulum, which contains an example that solves a system of higher-order nonlinear ODE.

Chapter 2 will cover how to represent the data of linear ODE in Drasil. Then, in Chapter 3, we will analyze external libraries. In Chapter 4, we will explore how to connect the Drasil ODE data representation with external libraries.

Chapter 2

ODE Data Representation

In the Drasil framework, there is a single data structure containing all the information for all products; we call it `System Information`. The giant `System Information` collects a multitude of information; whenever we need it, we extract the information from `System Information`. We store the ODE information in `System Information`.

An ODE can exist in various forms, with different forms suitable for different purposes. For example, we can transform a higher-order ODE into its equivalent system of first-order ODEs. The higher-order form is suitable for presenting the physics, but the system form is suitable for solving the ODE numerically. In previous research, we stored the ODE information in a data structure that cannot be reused for other purposes. We have to manually create the ODE in a specific form to satisfy a new goal each time. This is problematic because duplication cause problems for maintainability and traceability. In many projects, developers want to remove manually created code as much as possible because they are tedious and error-prone. For example, the FEniCS is a project for solving differential equations. While solving differential equations with the finite element method, two variables, A_T and l_T, are

needed and usually calculated by hand (227) (2). To avoid creating special-purpose code, FEniCS developers decided to solve the problem by creating a form complier. Another example is the ATLAS project, a software tool for solving linear algebra. Researchers can implement many optimizations in specific platforms, but those optimizations may cause a slowdown on other platforms (3) (16). A traditional way of handling this problem is manually creating code for each platform. To avoid creating manually created code, ATLAS researchers decided to create a new paradigm called Automated Empirical Optimization. The Drasil team faces a similar problem: we manually extract necessary information from the original ODE. The solution is to explore a new way to store ODEs in a new data structure. The new structure must allow the different forms to be isomorphic, which means we can map one ODE form to another.

By capturing the essential information of ODEs, we can flexibly transform it into others forms that might appear. On the one hand, the transformation might result in losing information. For example, we can display an ODE in a textual form in the SRS (software requirements specification). Once we complete the transformation for displaying purposes, the textual form ODE has less information than the original ODE. On the other hand, we may make a transformation for a specific purpose, such as generating code for an ODE solver. Furthermore, capturing an ODE's information in a flexible data structure only requires a Drasil user to write the ODE's information once.

In this chapter, we will first introduce how we can rewrite ODE for a different purpose. Then, we will introduce a new data structure for storing ODE information. We will talk about details on how the new data structure captures ODE information,

how to use the new data structure, and how the new data structure interacts with the Drasil Printer.

2.1 Explicit Equation

Before we conducted this project, the Drasil framework can generate software that provides numerical solutions for a first-order ODE by explicitly rewriting the ODE equation. We will rewrite the original ODE in a form which the Drasil Code Generator can utilize. The Drasil Code Generator retrieves ODE information from the new ODE form and generates a program that can produce numerical solutions. We will illustrate our steps with the NoPCM case study. Figure 2.1 shows how a NoPCM example works in a lab environment. The rectangle represents a water tank, and it

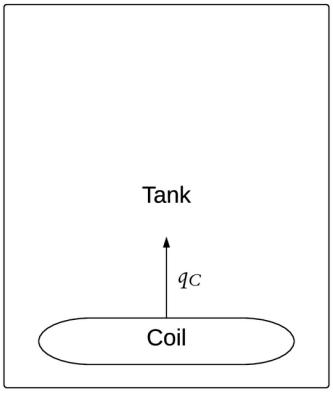

Figure 2.1: NoPCM Demonstration

has a heating coil at the bottom of the tank. qc represents the heat flux into the water from the coil. The goal is to estimate the temperature of the water over time. We can describe this natural phenomenon in a second-order ODE, Equation 2.1.1.

$$T'_w(t) + \frac{T_w(t)}{\tau_w} = \frac{T_c}{\tau_w} \tag{2.1.1}$$

$T_w(t)$ is a function of the independent variable, in this case time (t). T_w is the temperature of water ($°C$). $T'_w(t)$ is the first derivative of the function $T_w(t)$ with respect to time. T_c is the temperature of the heating coil (°C), and τ_w is the ODE parameter for water related to decay time (s). Isolating for $T'_w(t)$, we obtain the following equation:

$$T'_w(t) = \frac{1}{\tau_w}(T_c - T_w(t)) \tag{2.1.2}$$

```
-- Pseudocode for the readability
T_w'(t) = reciprocal τ_w * (T_c - T_w(t))
```

Code 2.1: NoPCM Equation for SRS

Based on Equation 2.1.2, we can encode the syntax of the ODE in Code 2.1 with the Drasil language. Although it has a basic mathematical structure, it is too hard-coded to make a transformation. We can use the information of Code 2.1 for a display purpose, such as displaying the ODE equation in the SRS, because this form follows the syntax of the equation. However, we cannot reuse it for other purposes, such as creating a program that solves the ODE numerically. Therefore, we rewrite the Code 2.1 to other forms for the purpose of solving the ODE. Brooks's thesis (91-103) (3) documented how the Drasil framework solves Equation 2.1.2 with the manually created ODEInfo. The `ODEInfo` is a `data type` that includes the necessary information from the original ODE (Equation 2.1.2). The Drasil Code Generator

can utilize `ODEInfo` to generate a program that solves the original ODE numerically. Code 2.2 shows how to rewrite the original ODE for the purpose of solving it. We

```
-- Pseudocode for the readability
reciprocal τ_w * (T_c - T_w[0])
```

Code 2.2: NoPCM Equation for the Drasil Code Generator

cannot directly transform Code 2.1 to Code 2.2; there is a gap between two ODE expressions. $T\_w(t)$ is just a variable, but $T\_w[0]$ means the index of 0 in T_w. This assumes T_w is a list.

Despite the gap between Code 2.1 and Code 2.2, we can manually close it by rewriting the ODE. Rewriting the ODE in another form produces duplication because both Code 2.1 and Code 2.2 describe the same ODE. The Code 2.1 lacks the necessary structure to allow transformation. Therefore, we propose a new data structure to store the ODE information.

2.2 Matrix Form Structure

In general, an equation contains a left-hand expression, a right-hand expression, and an equal sign. The left-hand and right-hand expressions connect by an equal sign. A linear ODE also has its left-hand and right-hand sides. Each side has its unique shape. We can write a linear ODE in the shape of

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.2.1}$$

On the left-hand side, $\boldsymbol{A}$ is an $m \times n$ matrix, and $\boldsymbol{x}$ is an n-vector. On the right-hand side, $\boldsymbol{b}$ is an m-vector. $\boldsymbol{A}$ is commonly known as the coefficient matrix, $\boldsymbol{x}$ is the unknown vector, and $\boldsymbol{b}$ is the constant vector. Equation 2.2.1 can represent not only a single linear ODE but also a linear system of ODEs. A linear system of ODEs is a finite set of linear differential equations. In this research, we have two case studies, NoPCM and PDController, for single higher-order linear ODEs and Double Pendulum for a system of higher-order nonlinear ODEs. The new data structure will be applied to single higher-order linear ODEs. This data structure is capable of storing information for a system of ODEs, but its related functions only support the case studies with a single ODE. Here is an ODE example from the PDContoller case study.

$$y''(t) + (1 + K_d) \cdot y'(t) + (20 + K_p) \cdot y(t) = r_t \cdot K_p \tag{2.2.2}$$

In Equation 2.2.2, there is only one dependent variable y. The dependent variable y is dependent on the independent variable t, in this case time. We use y(t) to represent a function of time. y'(t) is the first derivative of y(t). y''(t) is the second derivative of y(t). y is the process variable, and $y'(t)$ is the rate of change of $y(t)$. $y''(t)$ is the rate of change of the rate of change of $y(t)$. K_d, K_p, and r_t are constant variables. K_d is Derivative Gain, K_p is Proportional Gain, and r_t is the Set-Point. We can write Equation 2.2.2 in a matrix form as follows:

$$\begin{bmatrix} 1, & 1 + K_d, & 20 + K_p \end{bmatrix} \cdot \begin{bmatrix} y''(t) \\ y'(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} r_t \cdot K_p \end{bmatrix} \tag{2.2.3}$$

The relationship between the matrix form 2.2.1 and the Equation 2.2.3 is not hard to find. Firstly, the coefficient matrix $\boldsymbol{A}$ is a 1×3 matrix that consists of 1, $1 + K_d$, ane $20 + K_p$. Secondly, the unknown vector $\boldsymbol{x}$ is a 3×1 vector with $y''(t)$, $y'(t)$, and $y(t)$. Lastly, the constant vector $\boldsymbol{b}$ is a 1×1 vector with $r_t \cdot K_p$. The matrix form 2.2.1 captures all the knowledge we need to present an ODE. However, what is the matrix form for a nth-order linear ODE? Based on Paul's Online Notes (1), we can write all linear ODEs in the shape of

$$a_n(t) \cdot y^n(t) + a_{n-1}(t) \cdot y^{n-1}(t) + \cdots + a_1(t) \cdot y'(t) + a_0(t) \cdot y(t) = h(t) \tag{2.2.4}$$

The coefficient $a_0(t), \ldots, a_n(t)$ and $g(t)$ can be constant or non-constant functions, in our case they are constant functions. We also can write Equation 2.2.4 in a matrix form as

$$\begin{bmatrix} a_n(t), & a_{n-1}, \ldots, & a_0(t) \end{bmatrix} \cdot \begin{bmatrix} y^n(t) \\ y^{n-1}(t) \\ \cdots \\ y(t) \end{bmatrix} = \begin{bmatrix} h(t) \end{bmatrix} \tag{2.2.5}$$

This is the methodology used for linear ODEs, and it contains all the necessary information for understanding the linear ODE. Therefore, we create a data structure that contains the matrix information of the ODE. It is an advanced structural ODE information `data type`, called `DifferentialModel` to capture the knowledge of linear

ODEs.

The `DifferentialModel` is the type and takes one value. The `SystemOfLinearODEs` is a value with a record that is used to describe the structural content of a system of linear ODEs with six necessary fields. Here is the representing code for `DifferentialModel`.

```
data DifferentialModel = SystemOfLinearODEs {
  _indepVar :: UnitalChunk,
  _depVar :: ConstrConcept,
  _coefficients :: [[Expr]],
  _unknowns :: [Unknown],
  _dmConstants :: [Expr],
  _dmconc :: ConceptChunk
}
```

Previous to this research, UnitalChunk, ConstrConcept, Expr, and ConceptChunk already existed in Drasil. We created an `Unknown` type for this experiment. Their semantics will show up in Table 2.1

Type	**Semantics**
`UnitalChunk`	concepts with quantities that must have a unit definition.
`ConstrConcept`	conceptual symbolic quantities with Constraints and maybe a reasonable value.
`Expr`	a type to encode a mathematical expression.
`ConceptChunk`	a concept that contains an idea, a definition, and an associated domain of knowledge
`Unknown`	synonym of Integer

Table 2.1: Type Use in DifferentialModel

The `_indepVar` represents the independent variable, and it is often time. The `_depVar` represents the dependent variable. Combing `_depVar` and `_indepVar`, it represents a function produce dependent variables over time. The `_coefficients` is a list of lists `Expr`, and it represents the coefficient matrix $\boldsymbol{A}$. The `_unknowns` is a list of `Unknown`, and `Unknown` is a synonym of integers. The `_unknowns` represents the order of functions. Combining `_depVar`, `_indepVar` and `_unknowns`, they can represent the unknown vector $\boldsymbol{x}$. The `_dmConstants` is a list of `Expr`, and it represents the constant vector $\boldsymbol{b}$. Lastly, the `_dmconc` contains metadata of this model. To represent Equation 2.2.2 in `DifferentialModel`, `_indepVar` is time, t, `_depVar` is y, `_coefficients` is a 1×3 matrix, `_unknowns` is a 3×1 vector, `_dmConstants` is the 1×1 vector, and `_dmconc` is `ConceptChunk` that describes this model. Code 2.3 shows the internal data representation of Equation 2.2.2 in `DifferentialModel`.

```
_indepVar = t -- time
_depVar = y -- the dependent variable
_coefficients = [[1, 1 + K_d, 20 + K_p]]
_unknowns = [2, 1, 0] -- orders
_dmConstants = [r_t * K_p]
_dmconc = ... -- Drasil definition for chuck concept
```

Code 2.3: Internal Data Representation for Equation 2.2.2

Currently, the `DifferentialModel` only captures the knowledge of linear ODEs with one dependent variable, and it is a special case of the family of linear ODEs. Studying this special case will help the Drasil team better understand how to capture the knowledge of all ODEs and eventually lead to solving a system of linear ODE with multiple dependent variables. On top of that, there is one assumption: the `_coefficients` can only be functions of independent variable `_indepVar`, often

time. In other words, the `_coefficients` should not depend on the dependent variable `_depVar`. The code does not currently check to ensure that this assumption is satisfied.

2.3 Input Language

There are many reasons why we want to provide an input language for users to input ODE equations. One major reason is that it could be over complicated for users to input a single ODE in a matrix form. While inputting a single ODE, one obvious way is directly passing value to each record via constructors of `DifferentialModel`. The Code 2.3 shows how to encode Equation 2.2.2 in the `DifferentialModel`. However, it would not be so elegant to set a single ODE in the example because users have to extract the coefficient matrix $\boldsymbol{A}$, unknown vector $\boldsymbol{x}$ and constant vector $\boldsymbol{b}$ from the original equation manually. Once the coefficient matrix, unknown vector and constant vector are ready, we can set values into `_depVar`, `_coefficients`, `_unknowns`, and `_dmConstants` accordingly. This process is ideal when the ODE is a system of ODE, and it would be over-complicated for a user to do extraction for a single ODE. Therefore, we decided to create a helper function to ease this issue. On top of that, the Drasil printer will print a single ODE in the SRS with a more familiar "one line equation" form rather than the matrix form. Another advantage of having a helper function to input an ODE is that it can reduce human error and make sure the equation is well-formed. We call this helper function the input language. We now describe this input language.

The input language is inspired by a linear nth-order ODE. Based on Paul's Online Notes (1), we can write all linear ODEs in the shape of Equation 2.2.4. On the

left-hand side of Equation 2.2.4, the expression is a collection of terms. Each term consists of a coefficient function and a derivative of the function $y(t)$. With ideas of term, coefficient, and derivative, we create new data types to mimic the mathematical expression of a linear ODE. The following is the detail of the code for new data types and operators.

```
type Unknown = Integer
data Term = T{
  _coeff :: Expr,
  _unk :: Unknown
}
type LHS = [Term]

($^^) :: ConstrConcept -> Integer -> Unknown
($^^) _ unk' = unk'

($*) :: Expr -> Unknown -> Term
($*) = T

($+) :: [Term] -> Term -> LHS
($+) xs x  = xs ++ [x]
```

For the new type, `LHS`, which is a short name for the left-hand side, is a list of `Term`. This corresponds to the left-hand side is a collection of terms. Each `Term` has an `Expr` and `Unknown`. This corresponds to a term consisting of a coefficient and a

derivative of the function. Although `_unk` is an integer, combining `_unk`, `_depVar` and `_indepVar` we can get the derivative of the function. New operators are inspired by the linear Equation 2.2.4. The `$^^` operator takes a variable and an integer, and it represents the derivative of the function. For instance, in Equation 2.2.2, we can write y `$^^ 2` to represent $y''(t)$. One thing we want to notice here is that we store $y(t)$ in `_depVar` and `_indepVar`. The operator `$^^` will ignore the first parameter and store the second parameter in `_unknowns`. The reason to position a dummy variable before `$^^` is that this will maintain the whole input structure to be close to a linear ODE. The `$*` operator creates a term by combining a coefficient matrix and a derivative function. For instance, in Equation 2.2.2, we can write $(1 + K_d)$ `$ *` $(y$ `$^^ 1)` to represent $(1 + K_d) \cdot y'(t)$. Lastly, the `$+` operator will append all terms into a list. Let's write code (Code 2.4) for the example matrix form 2.2.2 in the newly introduced input language. The full detail of the input language for the PDController example is shown in Appendix A.1.

```
-- Pseudocode for the readability
-- left hand side = y_t'' + (1 + K_d)y_t' + (20 + K_p)y_t
-- right hand side = r_t K_p

lhs = [1 $* (y $^^ 2)]
  $+ (1 + K_d) $* (y $^^ 1)
  $+ (20 + K_p) $* (y $^^ 0)
rhs = r_t * K_p
```

Code 2.4: Input Language for Equation 2.2.2

2.4 Two Constructors

There are many ways to create the `DifferentialModel`. One most obvious way is to set each field directly by passing values in the constructor, and makeASystemDE constructor serves this role. We also designed another constructor, makeASingleDE, for users who want to use the input language to create a `DifferentialModel`.

For `makeASystemDE` constructor, a user can set the coefficient matrix, unknown vector, and constant vector by explicitly giving `[[Expr]]`, `[Unknown]`, and `[Expr]`. There will be several guards to check whether the inputs are well-formed, as follows:

1. The coefficient matrix and constant vector dimension need to match. The `_coefficients` is an $m \times n$ matrix, and `_dmConstants` is an m-vector. This guard makes sure they have the same m dimension. If the dimensions do not match, the Drasil framework will throw an error: "Length of coefficients matrix should be equal to the length of the constant vector".

2. The dimension of each row in the coefficient matrix and unknown vector need to match. The `_coefficients` uses a list of lists to represent an $m \times n$ matrix. It means each list in `_coefficients` will have the same length n, and `_unknowns` is an n-vector. Therefore, the length of each row in the `_coefficients` should equal the length of `_unknowns`. In the case of a size mismatch, the error message is:"The length of each row vector in coefficients needs to be equal to the length of the unknown vector".

3. The order of the unknown vector needs to be descending due to our design decisions. We have no control over what users will give to us, and there are infinite ways to represent a linear equation in the matrix form 2.2.1. We strictly ask users to input the unknown vector in descending order so that we can maintain the shape of a

normal form of a linear ODE. This design decision will simplify the implementation for solving a linear ODE numerically in Chapter 3. If there is a mismatch, the error message will say, "The order of the given unknown vector needs to be descending".

Code 2.5 shows how to directly set Equation 2.2.2's coefficient matrix, unknown vector, and constant vector. This example is made for the PDContoller case study.

```
1  -- Pseudocode for the readability
2  imPDRC :: DifferentialModel
3  imPDRC = makeASystemDE
4    time
5    opProcessVariable
6    coeffs = [[1, 1 + K_d, 20 + K_p]]
7    unknowns = [2, 1, 0]
8    constants = [r_t * K_p]
9    "imPDRC"
10   (nounPhraseSP
     ↪  "Computation of the Process Variable as a function of time")
11   EmptyS
```

Code 2.5: Explicitly Set Values for Equation 2.2.2 in DifferentialModel

The second constructor is called `makeASingleDE`. This constructor uses the input language to simplify the input of a single ODE. In `makeASingleDE`, we create the coefficient matrix, unknown vector, and constant vector based on restricted inputs. Contrasting to the `makeASystemDE`, users have to input the ODE by using the input language we designed. In the backend, `DifferentialModel` will extract useful information from the input language and generate the coefficient matrix and unknown vector. The constructor first creates a descending unknown vector base on the order of the ODE. Using Code 2.4 as an example, the order of the ODE is 2, so we will generate the unknown vector $[2, 1, 0]$. Then, based on the generated unknown vector,

we will search for the corresponding coefficient from the input language and form a matrix. The main advantage of this design decision is that we rely on the input language to provide the ODE in the correct format. While we allow users directly set values for `DifferentialModel`, we have no guarantee the format of the input is correct. With help from the input language, users can check for syntax errors. Code 2.4 shows how to use the input language to set Equation 2.2.2 in a matrix form. The full detail of how to use the input language to set the coefficient matrix, unknown vector, and constant vector for the PDContoller example is shown in Appendix A.1.

In Code 2.6, the `findHighestOrder` find the highest order n in a list of `Term`. Then, in `createAllUnknowns`, we create a list `Unknown` $[n, n-1, ..., 0]$ in descending order. This list is the `_unknowns` in `DifferentialModel`.

```
1  -- | Find the highest order in left hand side
2  findHighestOrder :: LHS -> Term
3  findHighestOrder = foldr1 (\x y -> if x ^. unk >= y ^. unk then x
   ↪  else y)
4
5  -- | Create all possible unknowns based on the highest order.
6  -- | The order of the result list is from the highest degree to
   ↪  zero degree.
7  createAllUnknowns :: Unknown -> ConstrConcept -> [Unknown]
8  createAllUnknowns highestUnk depv
9    | highestUnk  == 0  = [highestUnk]
10   | otherwise = highestUnk : createAllUnknowns (highestUnk - 1)
   ↪  depv
```

Code 2.6: Emulate Unknown

Code 2.7 demonstrate how to create `_coefficients` for `DifferentialModel`. We loop through the list of `[Unknown]`. Based on each individual `Unknown`, we can find its corresponding `Term` in a list of `Term`. We collect its `Expr`. If we did not find a

matched `Term`, we would use 0 as the `Expr`.

```haskell
-- | Create Coefficients base on all possible unknowns
-- | The order of the result list is from the highest degree to
 ↪  zero degree.
createCoefficients :: LHS -> [Unknown] -> [Expr]
createCoefficients [] _ = error "Left hand side is an empty list"
createCoefficients _ [] = []
createCoefficients lhs (x:xs) = genCoefficient (findCoefficient x
 ↪  lhs) : createCoefficients lhs xs

-- | Get the coefficient, if it is Nothing, return zero
genCoefficient :: Maybe Term -> Expr
genCoefficient Nothing = exactDbl 0
genCoefficient (Just x) = x ^. coeff

-- | Find the term that match with the unknown
findCoefficient :: Unknown -> LHS -> Maybe Term
findCoefficient u = find(\x -> x ^. unk == u)
```

Code 2.7: Create a Coefficient Matrix

2.5 Display Matrix

After a `DifferentialModel` obtains the ODE information, we want to display them in the SRS. Previously, we mentioned the Drasil framework is able to generate software artifacts, and the SRS is one of them. This section will discuss two ways to display ODEs in the SRS.

1. We can display ODEs in a matrix form. The matrix form 2.2.3 is the prototype of how the ODE will appear in a matrix form in the SRS. In the `DifferentialModel`, the `_coefficients` is a list of lists `Expr`, the unknown vector is a list of `Unknown`,

Equation

$$[1 \quad 1+K_\mathrm{d} \quad 20+K_\mathrm{p}] \cdot \begin{bmatrix} \frac{d^2 y_\mathrm{t}}{dt^2} \\ \frac{dy_\mathrm{t}}{dt} \\ y_\mathrm{t} \end{bmatrix} = [r_\mathrm{t} K_\mathrm{p}]$$

(a) Displaying ODE in a Matrix Form

Equation

$$\frac{d^2 y_\mathrm{t}}{dt^2} + (1+K_\mathrm{d})\frac{dy_\mathrm{t}}{dt} + (20+K_\mathrm{p}) y_\mathrm{t} = r_\mathrm{t} K_\mathrm{p}$$

(b) Displaying ODE as a Linear Equation

Figure 2.2: Options of Displaying an ODE

and the constant vector is a list of `Expr`. It should be fairly straightforward for Drasil Printer to display them by printing each part sequentially. Figure 2.2a shows how to display a matrix of ODEs in the SRS.

2. We also can display ODEs in a shape of a linear equation. Equation 2.2.2 is the prototype of how the ODE will show up in the shape of a linear equation in the SRS. Displaying a single ODE in a linear equation is a special case. When there is only one ODE, it would be over complicated to display it as a matrix. We explicitly force the Drasil printer to display a single ODE in the shape of a linear equation (Figure 2.2b). The example is just a demo that shows the Drasil printer is capable of displaying an ODE in a matrix form (Figure 2.2a).

In the future, the Drasil team wants to explore more variability in representing ODEs. An ODE has various forms, and we want `DifferentialModel` to represent as many forms as possible. One topic highlighted in the discussion is showing an ODE in a canonical form. However, many mathematicians have different opinions on a canonical form, and the name of canonical form has been used differently, such

as normal form or standard form. More research on this part would help us better understand the knowledge of ODE.

Chapter 3

External libraries

External libraries are from an outside source; they do not originate from the source project. Our current interest is for libraries that are used to support solving scientific problems. We use external libraries to solve the ODE to save resources. Creating a complete ODE solver in Drasil would take considerable time, and there are already many reliable external libraries have been tested by long use.

The external libraries are bodies of mathematical knowledge that are accessed through a well-defined interface. Since all external libraries are language-dependent, the Drasil framework needs to generate many interfaces to utilize external libraries. GOOL (Generic Object-Oriented Language) is the module to do this job. With the implementation of GOOL, the Drasil framework can generate five different languages: Python, Java, C++, C#, and Swift. Among those five languages, four programming languages have ODE libraries for solving ODEs; we did not find a suitable library for Swift. In Python, the Scipy library (14) is a well-known scientific library for solving scientific problems, including support for solving ODEs. In Java, a library called Apache Commons Maths (ACM) (7) provides a supplementary library for solving

mathematical and statistical problems not available in the Java programming language. ACM includes support to solve ODEs. Two less known libraries to solve ODEs are the ODEINT Library (10) in C++ and the OSLO Library (12) in C#. There could be multiple external libraries to solve the ODE in one language, but we currently only support one library for each language.

We believe it is beneficial to conduct a commonalty analysis for all four selected libraries because the Drasil framework is suited to generate program families. A program families (4) is a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. In this case, we may want to instruct the Drasil Code Generator to create programs that solve ODEs in multiple algorithms or allow other output types to interact with other modules. Those programs have parameterizable variabilities, so we can take advantage of developing them as a family (5).

The four selected libraries have commonalities and variabilities. Firstly, they all provide a numerical solution for a system of first-order ODEs. Each library can output a value of the dependent variable at a specific time, and we can collect those values in a time range. Secondly, they all provide different algorithms for solving ODEs numerically; we will conduct a rough commonality analysis of available algorithms in Section 3.2. A complete commonality analysis would be too time-consuming and out of the scope of our study. Lastly, the OSLO library have the potential to output an ODE in different types. This discovery will provide options for the Drasil framework to solve an ODE by generating a library rather than a standalone executable program.

This chapter will discuss topics related to the commonalities and variabilities of the four libraries, including numerical solutions, algorithms options and outputting

an ODE in different types.

3.1 Numerical Solutions

We use algorithms to make approximations for mathematical equations and create numerical solutions. All numerical solutions are approximations, and some numerical solutions that utilize better algorithms can produce better results than others. All selected libraries provide numerical solutions for a system of first-order ODEs as an IVP (Initial Value Problem). The IVP requires an initial condition that specifies the function's value at the start point, contrasting with a BVP (Boundary Value Problem). In a BVP, we apply boundary values instead of initial values. In this research, we will solve each scientific problem as an IVP. Let's see how to solve a system of first-order ODEs with an example.

The following example is derived from Equation 2.2.2. We transform the second-order ODE into a system of first-order ODEs. We replaced $y(t)$ with $x_1(t)$, and $y'(t)$ with $x_2(t)$. The details on how to convert a higher-order linear ODE to a system of first-order ODEs are shown in Section 4.1. At this point, our goal is to show an example of how we encode a system of first-order ODEs in Drasil.

$$\begin{aligned} x_1'(t) &= x_2(t) \\ x_2'(t) &= -(1 + K_d) \cdot x_2(t) - (20 + K_p) \cdot x_1(t) + r_t \cdot K_p \end{aligned} \tag{3.1.1}$$

In Equation 3.1.1, there are two dependent variables: x_1 and x_2. Both $x_1(t)$ and

$x_2(t)$ are functions of the independent variable, in this case time t. x_1 is the process variable, and x_2 is the rate of change of x_1. $x_1'(\mathrm{t})$ is the first directive of the function $x_1(t)$ with respect to time, and $x_2'(\mathrm{t})$ is the first derivative of the function $x_2(\mathrm{t})$ with respect to time. K_d, K_p, and r_t are constant variables; they have the same meaning as in Equation 2.2.2 and Equation 3.1.1. We can encode Equation 3.1.1 in all four libraries.

In Python Scipy library, we can write the example as follows:

```
1 def f(t, y_t):
2     return [y_t[1], -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0]
    ↪  + r_t * K_p]
```

In this example, y_t is a list of dependent variables. The index 0 of y_t is the dependent variable x_1, and the index 1 of y_t is the dependent variable x_2. $y\_t[1]$ represent the first equation $x_1'(t) = x_2(t)$ in Equation 3.1.1. The second value in the returned list $(-(1.0 + K\_d) * y\_t[1] + -(20.0 + K\_p) * y\_t[0] + r\_t * K\_p)$ represents the second equation, $x_2'(t) = -(1+K_d)\cdot x_2(t) - (20+K_p)\cdot x_1(t) + r_t \cdot K_p$, in Equation 3.1.1.

In Java ACM library, we can write the example using the following code:

```
1 public void computeDeriv(double t, double[] y_t, double[] dy_t) {
2     dy_t[0] = y_t[1];
3     dy_t[1] = -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) * y_t[0] + r_t
  ↪  * K_p;
4 }
```

In C++ ODEINT library, we can write the example as the following code:

```
void ODE::operator()(vector<double> y_t, vector<double> &dy_t,
↪ double t) {
    dy_t.at(0) = y_t.at(1);
    dy_t.at(1) = -(1.0 + K_d) * y_t.at(1) + -(20.0 + K_p) *
↪ y_t.at(0) + r_t * K_p;
}
```

In C# OSLO library, we can write the example as the following code:

```
Func<double, Vector, Vector> f = (t, y_t_vec) => {
    return new Vector(y_t_vec[1], -(1.0 + K_d) * y_t_vec[1] +
↪ -(20.0 + K_p) * y_t_vec[0] + r_t * K_p);
};
```

Once we capture the information of the system of ODEs, we have to give an initial condition for solving an ODE as an IVP. To solve Equation 3.1.1, we must provide the initial value for both x_1 and x_2. Overall, an ODE is a simulation, and it simulates a function of time. Before we start the simulation, other configurations need to be specified, including the start time, end time, and time step between each iteration. We can also provide values for each library's absolute and relative tolerance. Those two tolerances control the accuracy of the solution. As we mentioned before, all numerical solutions are approximations. High tolerances produce less accurate solutions, and smaller tolerances produce more accurate solutions. Lastly, we have to collect the numerical output for each iteration. The full details on how each library solves Equation 3.1.1 are shown in Appendix A.2 and Code 3.3.

3.2 Algorithm Options

We can solve an ODE with many algorithms. The four selected libraries each provide many algorithms. We roughly classify available algorithms into four categories based on the type of algorithm they use. They are a family of Adams methods, a family of backward differentiation formula methods (BDF), a family of Runge-Kutta (RK) methods, and a "catch all" category of other methods. The commonality analysis we provide on available algorithms is a starting point. It is an incomplete approximation. Getting a complete commonality analysis will require help from domain experts in ODE. Although the commonality is incomplete, the team still benefits from the current analysis. Not only can a future student quickly access information on which algorithm is available in each language, but also the analysis reminds us that we can increase the consistency of artifacts by providing one-to-one mapping for each algorithm in the four libraries. For example, if a user explicitly chooses a family of Adams methods as the targeted algorithm, all available libraries should use a family of Adams methods to solve the ODE. Unfortunately, not all libraries provide a family of Adams methods. The targeted algorithm will affect what languages we can generate. Table 3.1 shows the availability of a family of algorithms in each library. The full details of each library's algorithm availability are shown in Appendix A.3.

3.3 Output an ODE

In the Drasil framework, there is an option to generate modularized software. This modularized software currently contains a controller module, an input module, a calculation module, and an output module. The controller module contains the main

Library / Algorithm	**Scipy-Python**	**ACM-Java**	**ODEINT-C++**	**OSLO-C#**
Family of Adams	• Implicit Adams	• Adams Bashforth • Adams Moulton	• Adams Bashforth Moulton	
Family of BDF	• BDF			• Gear's BDF
Family of RK	• Dormand Prince (4)5 • Dormand Prince 8(5,3)	• Explicit Euler • 2ed order • 4th order • Gill fourth order • 3/8 fourth order • Luther sixth order • Higham and Hall 5(4) • Dormand Prince 5(4) • Dormand Prince 8(5,3)	• Explicit Euler • Implicit Euler • Symplectic Euler • 4th order • Dormand Prince 5 • Fehlberg 78 • Controlled Error Stepper • Dense Output Stepper • Rosenbrock 4 • Symplectic RKN McLachlan 6	• Dormand Prince RK547M
Others		• Gragg Bulirsch Stoer	• Gragg Bulirsch Stoer	

Table 3.1: Algorithms Support in External Libraries

function that starts the software. The input module handles all input parameters and constraints. We manually create a text file that contains all input information. For example, in Double Pendulum, the input module will read Code 3.1 and convert the information to its environment.

```
# Length of the upper rod (m)
2.0
# Length of the bottom rod (m)
1.0
# Mass of the upper object(kg)
0.5
# Mass of the bottom object(kg)
2.0
```

Code 3.1: A Sample Input File for Double Pendulum

The calculation module contains all the logic for solving the scientific problem. For example, in Double Pendulum, the calculation module contains all functions for calculating the numerical solution. Lastly, the output module will output the solution. In all ODE case studies, the output module will write the values returned by the calculation module as a string. For example, in Double Pendulum, the output module write Code 3.2 in a text file.

```
# this is theta 1
theta = [1.3463968515384828, 1.3463947169563892,
↪  1.346388313227267, 1.3463776404025904, 1.3463626985681507,
↪  1.3463434878440559, ... ]
```

Code 3.2: A Output File for Double Pendulum

With each module interacting with others we would like to study the output of

the calculation module in the ODE case studies. Currently, the calculation module will output a finite sequence of real numbers, $\mathbb{R}^m$, for example, a list of numbers in Python. m is a natural number which depends on the start time, end time, and time step. We have the following specification for the calculation module:

Module Name	**Input**	**Output**
Calculations	$\mathbb{R}^n$	$\mathbb{R}^m$

Table 3.2: Specification for Calculations Module Returns a Finite Sequence

$\mathbb{R}^n$ represents input values, and the superscript n means how many input values. In our case study, after running the generated program, it will create a file containing the numerical solution of the ODE from the start time to the end time. The numerical solution is written as a stream of real numbers in Code 3.2. A finite sequence of real numbers only captures a partial solution; we ideally want to capture a complete solution. Therefore, we would like to explore options to output a different type for the calculation module.

Most selected external libraries only provide numerical solutions in the form of a finite sequence of real numbers, $\mathbb{R}^m$. The C# OSLO library not only supports outputting a finite sequence of real numbers but also an infinite sequence of real numbers (we use `double` as the type for real numbers in C# code). In C# OSLO library, we can get an infinite numerical solution that contains all possible values of the dependent variable over time ($\mathbb{R}^\infty$). ∞ is the length of the sequence, and it is a natural number. The function `Ode.RK547M` returns an endless enumerable sequence of solution points. If we are interested in a partial solution ($\mathbb{R}^m$), we can filter it with parameters such as start time, end time, and time step. Code 3.3 shows the full details of how to solve Equation 3.1.1 in the OSLO library.

```
1  public static List<double> func_y_t(double K_d, double K_p, double
   ↪  r_t, double t_sim, double t_step) {
2      List<double> y_t;
3      Func<double, Vector, Vector> f = (t, y_t_vec) => {
4          return new Vector(y_t_vec[1], -(1.0 + K_d) * y_t_vec[1] +
   ↪  -(20.0 + K_p) * y_t_vec[0] + r_t * K_p);
5      };
6      Options opts = new Options();
7      opts.AbsoluteTolerance = Constants.AbsTol;
8      opts.RelativeTolerance = Constants.RelTol;
9
10     Vector initv = new Vector(new double[] {0.0, 0.0});
11     IEnumerable<SolPoint> sol = Ode.RK547M(0.0, initv, f, opts);
12     IEnumerable<SolPoint> points = sol.SolveFromToStep(0.0, t_sim,
   ↪  t_step);
13     y_t = new List<double> {};
14     foreach (SolPoint sp in points) {
15         y_t.Add(sp.X[0]);
16     }
17
18     return y_t;
19 }
```

Code 3.3: Source Code of Solving PDController in OSLO

In Code 3.3, between line 3 and line 4, we encode the ODE of Equation 3.1.1 in a `Func`. Between line 7 and line 8, we set the absolute and relative tolerance in the `Options` class. In line 10, we initialize initial values. Next, in line 11, we use `Ode.RK547M` to get an endless sequence of real numbers, $\mathbb{R}^{\infty}$. In line 12, we use `SolveFromToStep` to get a partial solution ($\mathbb{R}^{m}$) based on the start time, the final time, and the time step. Last, between line 13 and line 15, we run a loop to collect the process variable x_1. With the workflow we described above, the `Ode.RK547M(0.0, initv, f, opts)` returns an object with richer data because

$\mathbb{R}^m \subset \mathbb{R}^\infty$. Instead of returning $\mathbb{R}^m$, we can have an option to return $\mathbb{R}^\infty$. Here is the new specification.

Module Name	**Input**	**Output**
Calculations	$\mathbb{R}^n$	$\mathbb{R}^\infty$

Table 3.3: Specification for Calculations Module Return an Infinite Sequence

The implementation of this specification is not complete, but we provide an analysis of what options the C# OSLO library offers.

Ideally, the ODE is a function that means giving an independent variable will output dependent variables. Here is another proposed specification:

Module Name	**Input**	**Output**
Calculations	$\mathbb{R}^n$	$\mathbb{R} \to \mathbb{R}^k$

Table 3.4: Specification for Calculations Return a Funtion

In output $\mathbb{R} \to \mathbb{R}^k$, $\mathbb{R}$ is the independent variable, and $\mathbb{R}^k$ is a sequence that contains dependent variables. For a fourth-order ODE, $\mathbb{R}^k$ would be $\mathbb{R}^4$. Since Drasil Framework can generate a library, the idea of outputting an ODE as a function can be useful. A program can hook up the interface of the generated library, and the library will provide support for calculating the numerical solution of the ODE. The implementation of this specification is not complete, but it gives future students some inspiration on how to generate a library to solve the ODE in Drasil. We defined how to generate the code for external libraries in Drasil Code Generator. To generate new interfaces for each library in each language, future students need to write new instructions.

Chapter 4

Connect Model to Libraries

We store the information of a higher-order linear ODE in the `DifferentialModel` in Chapter 2. This `data type` captures the structure of the ODE so that we can transform the ODE into other forms. Chapter 3 discusses how to solve a system of first-order ODEs numerically in four selected external libraries. However, there is a gap between the `DifferentialModel` and external libraries. The selected libraries cannot solve the higher-order ODE directly, but they can solve its equivalent system of first-order equations. We know that most ways of solving ODEs are intended for systems of first-order ODEs, so we want to convert the higher-order ODE to a system of first-order ODEs (17). Firstly, we transform a linear higher-order ODE into a system of first-order ODEs. Then, we use the Drasil Code Generator to generate code that contains proper interfaces for utilizing four selected external libraries. This program solves the system of first-order ODEs numerically by producing a list of values of dependent variables based on time. The original linear higher-order ODE is equivalent to the system of first-order linear ODEs we solve. Thus, by transitivity, the numerical solution for the system of first-order ODEs is also the numerical solution

of the higher-order ODE.

In this research, my work primarily focuses on enabling the Drasil Code Generator to generate code for solving higher order ODE and automatically extracting useful information from `DifferentialModel` for the Drasil Code Generator in single higher-order linear ODEs. In addition, we create a new case study, Double Pendulum, which has a system of nonlinear ODEs. We handle the nonlinear ODE differently than the linear ODE. The nonlinear ODE still requires Drasil users manually extract information from the original ODE. With the new implementations, we can generate code to solve the Double Pendulum numerically. In this chapter, we will first discuss how to convert any higher-order linear ODE to a system of first-order ODEs in theory. Then, we will discuss how to enable the Drasil Code Generator to generate code that produces a numerical solution for a system of first-order ODEs. Lastly, we will discuss how to automate the generation process.

4.1 Higher Order to First Order

Any higher-order linear ODE can be written in the following form:

$$y^n = f(t, y, y', y'', \ldots, y^{(n\text{-}1)}) \tag{4.1.1}$$

We isolate the highest derivative y^n on the left-hand side and move the rest of the terms to the right-hand side. On the right-hand side, $f(t, y, y', y'', \ldots, y^{(n\text{-}1)})$ means a function depends on variables t, y, y', $\ldots$, and $y^{(n\text{-}1)}$. t is the independent variable, often time. y, y', $\ldots$, and $y^{(n\text{-}1)}$ represent the dependent variable y, the first derivative of y, $\ldots$, up to $(n-1)^{\text{th}}$ derivative.

For the next step, we introduce the following new variables: x_1, x_2, ..., x_n. The number of newly introduced dependent variables equals the order of the ODE, n. The new relationship is shown below:

$$\begin{aligned} x_1 &= y \\ x_2 &= y' \\ &\cdots \\ x_n &= y^{(n\text{-}1)} \end{aligned} \tag{4.1.2}$$

Next, we differentiate x_1, x_2, ..., x_n in Equation 4.1.2 to establish the following relationships between the variables:

$$\begin{aligned} x_1' &= y' = x_2 \\ x_2' &= y'' = x_3 \\ &\cdots \\ x_{n-1}' &= y^{(n\text{-}1)} = x_n \\ x_n' &= y^n = f(t, x_1, x_2, \ldots, x_n) \end{aligned} \tag{4.1.3}$$

Since the higher-order ODE is a linear ODE, $f(t, x_1, x_2, \ldots, x_n)$ is a linear function; therefore, we can rewrite $f(t, x_1, x_2, \ldots, x_n)$ as

$$b_0(t) \cdot x_1 + b_1(t) \cdot x_2 + \cdots + b_{n-1}(t) \cdot x_n + h(t) \tag{4.1.4}$$

where $b_0(t)$, ..., $b_{n-1}(t)$ and $h(t)$ are constant functions or non-constant functions.

Based on Equation 4.1.3 and Equation 4.1.4, we obtain:

$$\begin{aligned} x_1' &= x_2 \\ x_2' &= x_3 \\ &\dots \\ x_{n-1}' &= x_n \\ x_n' &= b_0(t) \cdot x_1 + b_1(t) \cdot x_2 + ... + b_{n-1}(t) \cdot x_n + h(t) \end{aligned} \tag{4.1.5}$$

We can rewrite Equation 4.1.5 in a general form:

$$\boldsymbol{x'} = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{c} \tag{4.1.6}$$

The $\boldsymbol{A}$ is a coefficient matrix, and $\boldsymbol{c}$ is a constant vector. The $\boldsymbol{x}$ is the unknown vector that contains functions of the independent variable, often time. The $\boldsymbol{x'}$ is a vector that consists of the first derivatives of $\boldsymbol{x}$. The following is the long matrix form:

$$\begin{bmatrix} x_1' \\ x_2' \\ \dots \\ x_{n-1}' \\ x_n' \end{bmatrix} = \begin{bmatrix} 0, & 1, & 0, & \dots & 0 \\ 0, & 0, & 1, & \dots & 0 \\ \dots & & & & \\ 0, & 0, & 0, & \dots & 1 \\ b_0(t), & b_1(t), & b_2(t), & \dots & b_{n-1}(t) \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_{n-1} \\ x_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \dots \\ 0 \\ h(t) \end{bmatrix} \tag{4.1.7}$$

4.2 Connect Drasil with External Libraries

In previous research, Drasil developers wrote the ODE as a RelationConcept `data type`, which has a field called `_rel`. The `_rel` is ModelExpr type and can be used for representing mathematical expressions. We can write mathematical expressions such as the right-hand side equal to the left-hand side in `ModelExpr`. The drawback is that we cannot extract useful information from `RelationConcept` to allow the Drasil Code Generator to utilize that information. The Drasil printer can print a `RelationConcept` in the SRS, but the Drasil Code Generator cannot utilize the `RelationConcept`. Therefore, in the previous approach, the user has to repeat information through a manually created `data type`, called ODEInfo, to generate code. Our improved approach expanded the Drasil Code Generator's capability to generate code for solving a higher-order ODE.

4.2.1 Enable Drasil for Solving Higher-Order ODEs

The Drasil Code Generator utilized `ODEInfo` to generate code which produces a numerical solution. We can find details on how to generate code that solves a first-order ODE numerically in Brooks's thesis (91-103) (3). His work could generate code for a first-order ODE but not a higher-order ODE. A McMaster student, Naveen, created the PDContoller. It generates code for solving a second-order linear ODE numerically in Python only. The new approach will generate code for solving any higher-order linear ODE in Python, Java, C++ and C#.

Before our changes, `ODEInfo` only had an option to provide one initial value. For a higher-order ODE, the current setting of `ODEInfo` does not hold all information we need. `ODEInfo` needs to store multiple initial values to enable the Drasil Code

Generator to work for the higher-order ODE. For example, we need four initial values when we solve a fourth-order ODE as an IVP. Thus, the Drasil Code Generator must adapt to handle multiple initial values.

```
-- Old
data ODEInfo = ODEInfo {
  ...
  initVal :: CodeExpr
  ...
}

-- New
data ODEInfo = ODEInfo {
  ...
  initVal :: [CodeExpr],
  ...
}
```

Code 4.1: Source Code for Initial Values in Drasil

Code 4.1 changes the `data type` of `initVal` from a `CodeExpr` to a list of `CodeExpr`. Users can only set one initial value in the old way, but now they can set a list of multiple initial values. This change allows Drasil users to store multiple initial values in a list. We also have to ensure that Drasil Code Generator can utilize the new `data type [CodeExpr]`. Previously, Drasil Code Generator only handled the `initVal` as `CodeExpr`. Now the `initVal` becomes `[CodeExpr]`. In the Drasil framework, we handle a list of `data type` by matrix (In Drasil the `data type matrix` is used to represent both a 1D matrix (conventionally called a vector) and a 2D matrix.), and the code `initVal info` retrieves the `initVal` from the `ODEInfo data type`. The `matrix` can wrap a `[CodeExpr]` into a `CodeExpr`.

In `initSolListWithValFill`, `basicArgFill`, and `CodeDefinition` for `ODEInfo`,

```
matrix[initVal info]
```

```
# Old
  r.set_initial_value(T_init, 0.0)
  T_W = [T_init]

# New
  r.set_initial_value([T_init], 0.0)
  T_W = [[T_init][0]] # Initial values are also a part of the
   ↪  numerical solution, so we have to add the proper initial
   ↪  value to the list.
```

Code 4.2: Source Code for Initial Values in Python

we have to ensure we wrap the `initVal`.

The change of `basicArgFill` impacts generated code in the Python Scipy library and C# OSLO library. For the Python Scipy library, in Code 4.2, line 6 sets the initial value with a list of `T_init`.

The same thing happens in C# OSLO. In Code 4.3, line 5 initializes a list `T_init`, and we later use it as the initial values.

```
// Old
Vector initv = new Vector(T_init);

// New
Vector initv = new Vector(new double[] {T_init});
```

Code 4.3: Source Code for Initial Values in C#

In the Java ACM library, we use the FirstOrderDifferentialEquations interface to solve a system of first-order ODEs. This interface has a method called `getDimension()`.

```
1  // Old
2  public class ODE implements FirstOrderDifferentialEquations {
3    ...
4    public int getDimension() {
5      return 1;
6    }
7  }
8  // New
9  public class ODE implements FirstOrderDifferentialEquations {
10   ...
11   public int getDimension() {
12     return n; // n is an integer that users can explicitly define
13   }
14 }
```

Code 4.4: Source Code for Returning Dimension in Java

In the previous implementation, the `getDimension()` will always return 1 because we only solve a first-order ODE. It has been hard-coded in the Drasil Code Generator. Since we want to solve a n^{th}-order ODE, the `getDimension()` needs to return n.

To allow `getDimension()` to return an integer based on the order of the ODE, we have to add two additional methods based on function `fixedReturn`(100) (3) and fixedStatementFill(89) (3) in the Drasil Code Generator. The `fixedReturn` indicates a step that returns a fixed value, such as `fixedReturn (int 1)`. We create similar methods `fixedReturn'` and `fixedStatementFill'`. In Code 4.5, the `fixedReturn'` will indicate a step that returns a fixed value, but the `fixedStatementFill'` will fill in the fixed value base on the order of the ODE. Now, in Code 4.4, line 12, n will depends on the value we pass in `fixedStatementFill'`.

In C++, the backend code already handles the initial value as a list, so there is no change for artifacts.

```haskell
-- Old
fixedReturn :: CodeExpr -> Step
fixedReturn = lockedStatement . FRet

fixedStatementFill :: StepFill
fixedStatementFill = StatementF [] []

-- New added
fixedReturn' = statementStep (\cdchs [e] -> case (cdchs, e) of
  ([], _) -> FRet e
  (_,_) -> error
  ↪  "Fill for fixedReturn' should provide no CodeChunk")

fixedStatementFill' :: CodeExpr -> StepFill
fixedStatementFill' a = StatementF [] [a]
```

Code 4.5: Source Code for Returning a Fixed Value

Allowing multiple initial values unlocks the potential for Drasil to generate code that produces the numerical solution for a system of first-order ODEs. Every higher-order linear ODE has its equivalent system of first-order ODEs, and the solution for the system of first-order ODEs is also the solution for the higher-order ODE. The same thing happens on nonlinear higher-order ODEs. If we can transform a higher-order nonlinear ODE into a system of first-order ODEs, we can solve it via the four selected external libraries. For nonlinear ODEs, the process is not so automated; the user will need to do some manual work, as explained through the Double Pendulum example.

4.2.2 Double Pendulum

Figure 4.1 demonstrates how a double pendulum works in a lab environment. The full details of the Double Pendulum's SRS are located on the Drasil website. Table 4.1 lists all variables in the Double Pendulum example.

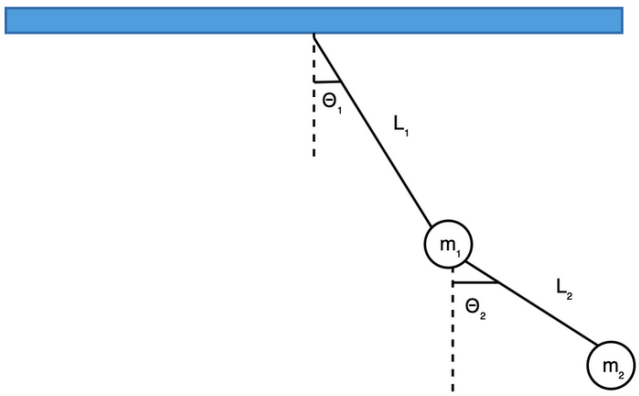

Figure 4.1: Double Pendulum Demonstration

Name	Description
$\mathtt{m}_1$	The mass of the first object
$\mathtt{m}_2$	The mass of the second object
$\mathtt{L}_1$	The length of the first rod
$\mathtt{L}_2$	The length of the second rod
$\mathtt{g}$	Gravitational acceleration
θ_1	The angle of the first rod
θ_2	The angle of the second rod
ω_1	The angular velocity of the first object
ω_2	The angular velocity of the second object

Table 4.1: Variables in Double Pendulum Example

At this point, we have primarily focused on solving linear ODEs. Despite the Double Pendulum case study contains a nonlinear ODE, the Drasil framework can still generate code to solve it numerically with some additional help from the user.

In the Double Pendulum case study, we want to solve the following equations:

$$\theta_1'' = \frac{-g(2m_1 + m_2)\sin\theta_1 - m_2 g\sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2)m_2({\theta_2'}^2 L_2 + {\theta_1'}^2 L_1\cos(\theta_1 - \theta_2))}{L_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))} \tag{4.2.1}$$

$$\theta_2'' = \frac{2\sin(\theta_1 - \theta_2)({\theta_1'}^2 L_1(m_1 + m_2) + g(m_1 + m_2)\cos\theta_1 + {\theta_2'}^2 L_2 m_2\cos(\theta_1 - \theta_2))}{L_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}$$

There are two second-order ODEs in one system. To solve this system of ODEs, we convert them into a system of first-order ODEs. The transformation follows the methodology we discussed in Section 4.1. We transform Equation 4.2.1 into Equation 4.2.2. Once the transformation is complete, we can encode Equation 4.2.2 and pass it to the Drasil Code Generator. However, we cannot show Equation 4.2.2 in the shape of Equation 4.1.6 because the ODE is not a linear ODE.

$$\theta_1' = \omega_1 \tag{4.2.2}$$

$$\theta_2' = \omega_2$$

$$\omega_1' = \frac{-g(2m_1 + m_2)\sin\theta_1 - m_2 g\sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2)m_2(\omega 2^2 L_2 + \omega 1^2 L_1\cos(\theta_1 - \theta_2))}{L_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}$$

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

Now that we have obtained Equation 4.2.2, we can encode it in Drasil. Code 4.6 shows the example of how we encode Equation 4.2.2 in Drasil. We replace the Drasil language with mathematical symbols to increase the readability of the code.

```
dblPenODEInfo :: ODEInfo
dblPenODEInfo = odeInfo
...
[3*π/7, 0, 3*π/4, 0] -- initial values
[ ω₁,
  -g(2m₁ + m₂)sin θ₁ - m₂gsin (θ₁ - 2θ₂) - 2sin (θ₁ - θ₂)m₂(ω₂²L₂ +
↪  ω₁²L₁cos (θ₁ - θ₂)) / L₁(2m₁ + m₂ -m₂cos (2θ₁ - 2θ₂)),
  ω₂,
  2sin (θ₁ - θ₂)(ω₁²L₁(m₁ + m₂ ) + g(m₁ + m₂ )cos θ₁ + ω₂²L₂m₂cos (θ₁
↪  - θ₂ )) / L₂(2m₁ + m₂ -m₂cos (2θ₁ - 2θ₂))
]
...
```

Code 4.6: Pseudocode for Encoding Double Pendulum's Equation

Once the `dblPenODEInfo` is ready, we will pass it to the Drasil Code Generator. It will generate code to solve the Double Pendulum in four languages. The details of the generated code are in Appendix A.4. However, the Double Pendulum case study cannot utilize any function introduced in the next section because they were designed for single linear ODEs.

The limitation of manually creating `ODEInfo` is that we will write the ODE twice in different ways. In this case, we encode both Equation 4.2.1 and Equation 4.2.2 in Drasil. They both demonstrate the same phenomenon and exist in an isomorphic ODE type. The following section will discuss how to automate the transformation from a higher-order linear ODE to a system of first-order ODEs.

4.3 Generate ODEInfo Automatically

Manually creating explicit equations is not ideal because it requires human interference and propagates duplicate information. We want to design the Drasil framework to be as fully automatic as possible. Therefore, an ideal solution is to encode the ODE in a flexible data structure. Then, we can extract information from this structure and generate a form of the ODE that selected external libraries can utilize. Creating the `DifferentialModel` data structure satisfies the need of this idea for linear ODEs. We can restructure an ODE based on the information from `DifferentialModel`. This research's scope only covers generating explicit equations for a single higher-order linear ODE. In the future, we want to generate explicit equations for a system of higher-order ODEs and nonlinear ODEs.

Once we encode the ODE in `DifferentialModel`, we want to restructure its equivalent system of first-order ODEs in the shape of Equation 4.1.6. For the convenience of implementation, we shuffle the data around in Equation 4.1.7. We reverse the order of $\boldsymbol{x}$ to $x_n, \ldots, x_1$. The coefficient matrix $\boldsymbol{A}$ also changes, but $\boldsymbol{x'}$ and $\boldsymbol{c}$ remain unchanged.

$$\begin{bmatrix} x_1' \\ \vdots \\ x_{n-2}' \\ x_{n-1}' \\ x_n' \end{bmatrix} = \begin{bmatrix} 0, & 0, & \cdots, & 1, & 0 \\ \vdots & & & & \\ 0, & 1, & \cdots, & 0, & 0 \\ 1, & 0, & \cdots, & 0, & 0 \\ b_{n-1}(t), & b_{n-2}(t), & \cdots, & b_1(t), & b_0(t) \end{bmatrix} \cdot \begin{bmatrix} x_n \\ \vdots \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ h(t) \end{bmatrix} \tag{4.3.1}$$

Since Equation 4.3.1 is an expansion of Equation 4.1.6, we will use symbols in

both equations to explain how to generate Equation 4.3.1. We highlighted $\boldsymbol{x'}$ and $\boldsymbol{x}$ in yellow in Equation 4.3.1. The number of elements in $\boldsymbol{x'}$ and $\boldsymbol{x}$ depends on how many new dependent variables are introduced. If the higher-order ODE is second-order, we will introduce two new dependent variables. If the higher-order ODE is n^{th}-order, we will introduce n new dependent variables. For $\boldsymbol{x'}$, knowing it is n^{th}-order ODE, we parameterize $x'_1, \ldots, x'_n$. For $\boldsymbol{x}$, knowing it is n^{th}-order ODE, we parameterize $x_n, \ldots, x_1$.

We highlighted the $n \times n$ coefficient matrix $\boldsymbol{A}$ in orange and blue in Equation 4.3.1. The orange part is a matrix that looks like the identity matrix, only rotated and with an extra column of 0s. For the lowest higher-order ODE, a second-order ODE, the orange part is $[1, 0]$. Equation 4.3.2 shows a completed transformation for a second-order linear ODE.

$$\begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} 1, & 0 \\ b_1(t), & b_0(t) \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ h(t) \end{bmatrix} \tag{4.3.2}$$

The $\boldsymbol{A}$ will be a 4×4 matrix for a fourth-order ODE. Equation 4.3.3 shows a completed transformation for a fourth-order ODE.

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \end{bmatrix} = \begin{bmatrix} 0, & 0, & 1, & 0 \\ 0, & 1, & 0, & 0 \\ 1, & 0, & 0, & 0 \\ b_3(t), & b_2(t), & b_1(t), & b_0(t) \end{bmatrix} \cdot \begin{bmatrix} x_4 \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ h(t) \end{bmatrix} \tag{4.3.3}$$

The orange part starts at the $(n-1)^{th}$ row with $[1, 0, \ldots]$. If there is a second row, we add $[0, 1, \ldots]$ above the start row and so on. We observe a pattern for the

```haskell
-- | Add Identity Matrix to Coefficients
-- | len is the length of the identity row,
-- | index is the location of identity value (start with 0)
addIdentityCoeffs :: [[Expr]] -> Int -> Int -> [[Expr]]
addIdentityCoeffs es len index
  | len == index + 1 = es
  | otherwise = addIdentityCoeffs (constIdentityRowVect len index
  ↪  : es) len (index + 1)

-- | Construct an identity row vector.
constIdentityRowVect :: Int -> Int -> [Expr]
constIdentityRowVect len index = addIdentityValue index $
  ↪  replicate len $ exactDbl 0

-- | Recreate the identity row vector with identity value
addIdentityValue :: Int -> [Expr] -> [Expr]
addIdentityValue n es = fst splits ++ [exactDbl 1] ++ tail (snd
  ↪  splits)
  where splits = splitAt n es
```

Code 4.7: Source Code for Creating Identity Matrix (Highlighted in Orange)

orange part so that we can generate it. In Code 4.7, `constIdentityRowVect` and `addIdentityValue` are responsible for generating each row in the orange part. We first create a row containing a list of 0. Then, we replace one of the 0s with 1. The `addIdentityCoeffs` runs through a recursion that combines all rows to form the orange and blue parts.

We highlighted the constant vector $\boldsymbol{c}$ in gray and red colour in Equation 4.3.1. The vector $\boldsymbol{c}$ has the length n. The last element of the constant vector $\boldsymbol{c}$ will be $h(t)$, and anything above $h(t)$ will be 0. In Code 4.8, in `addIdentityConsts`, given the expression of $[h(t)]$ and the order number of the ODE, we add $(n-1)$ 0s above the $h(t)$.

```haskell
-- | Add zeros to Constants
-- | len is the size of new constant vector
addIdentityConsts :: [Expr] -> Int -> [Expr]
addIdentityConsts expr len = replicate (len - 1) (exactDbl 0) ++
 ↪  expr
```

Code 4.8: Source Code for Creating Constant Matrix

The blue and red parts in Equation 4.3.1 can be determined by Equation 2.2.4. The `DifferentialModel` captures the relationship for Equation 2.2.4 but does not isolate the highest order to the left-hand side. To isolate the highest order, we have to shuffle terms between the left-hand side and right-hand side. The following is a demonstration of how to create the blue part. Giving the following equation:

$$a_n(t) \cdot y^n(t) + a_{n-1}(t) \cdot y^{(n\text{-}1)}(t) + \cdots + a_1(t) \cdot y'(t) + a_0(t) \cdot y(t) = h(t)$$

Firstly, we move every term from left to right, except the highest order term.

$$a_n(t) \cdot y^n(t) = -a_{n-1}(t) \cdot y^{(n\text{-}1)}(t) + \cdots + -a_1(t) \cdot y'(t) + -a_0(t) \cdot y(t) + h(t)$$

Secondly, we divide both sides of the equation by the coefficient $a_n(t)$.

$$y^n(t) = \frac{-a_{n-1}(t) \cdot y^{(n\text{-}1)}(t) + \cdots + -a_1(t) \cdot y'(t) + -a_0(t) \cdot y(t) + h(t)}{a_n(t)}$$

Then, we can write this in matrix form as:

$$\begin{bmatrix} y^n(t) \end{bmatrix} = \begin{bmatrix} -\frac{a_{n-1}(t)}{a_n(t)}, \ldots, & -\frac{a_1(t)}{a_n(t)} & -\frac{a_0(t)}{a_n(t)} \end{bmatrix} \cdot \begin{bmatrix} y^{(n-1)}(t) \\ \cdots \\ y'(t) \\ y(t) \end{bmatrix} + \begin{bmatrix} \frac{h(t)}{a_n(t)} \end{bmatrix}$$

Since $x'_n = y_n$ (Equation 4.1.3), we can replace y_n with x'_n. Based on Equation 4.1.2, we replace all derivatives of $y(t)$ with $x_n, \ldots, x_1$.

$$\begin{bmatrix} x'_n \end{bmatrix} = \begin{bmatrix} -\frac{a_{n-1}(t)}{a_n(t)}, \ldots, & -\frac{a_1(t)}{a_n(t)} & -\frac{a_0(t)}{a_n(t)} \end{bmatrix} \cdot \begin{bmatrix} x_n \\ \cdots \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} \frac{h(t)}{a_n(t)} \end{bmatrix}$$

Lastly, we replace new variables in Equation 4.3.1 to get a new matrix.

$$\begin{bmatrix} x'_1 \\ \cdots \\ x'_{n-2} \\ x'_{n-1} \\ x'_n \end{bmatrix} = \begin{bmatrix} 0, & 0, & \cdots, & 1, & 0 \\ \cdots & & & & \\ 0, & 1, & \cdots, & 0, & 0 \\ 1, & 0, & \cdots, & 0, & 0 \\ -\frac{a_{n-1}(t)}{a_n(t)}, & -\frac{a_{n-2}(t)}{a_n(t)}, & \cdots, & -\frac{a_1(t)}{a_n(t)}, & -\frac{a_0(t)}{a_n(t)} \end{bmatrix} \cdot \begin{bmatrix} x_n \\ \cdots \\ x_3 \\ x_2 \\ x_1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ \frac{h(t)}{a_n(t)} \end{bmatrix} \quad (4.3.4)$$

Here is the implementation for creating Equation 4.3.4 in Drasil. In Code 4.9, we remove the highest order because we want to isolate the highest order to the left-hand

side.

```haskell
-- | Delete the highest order
transUnknowns :: [Unknown] -> [Unknown]
transUnknowns = tail
```

Code 4.9: Source Code for Isolating the Highest Order

In Code 4.10, the `transCoefficients` divide the coefficient $a_n(t)$ in blue highlighted. The `divideConstants` divide the coefficient in the red highlights.

```haskell
-- | Cancel the leading coefficient of the highest order in the
 ↪  coefficient matrix
transCoefficients :: [Expr] -> [Expr]
transCoefficients es
  | head es == exactDbl 1 = mapNeg $ tail es
  | otherwise = mapNeg $ tail $ map ($/ head es) es
    where mapNeg = map (\x -> if x == exactDbl 0 then exactDbl 0
 ↪  else neg x)

-- | divide the leading coefficient of the highest order in
 ↪  constant
divideConstants :: Expr -> Expr -> Expr
divideConstants a b
  | b == exactDbl 0 = error "Divisor can't be zero"
  | b == exactDbl 1 = a
  | otherwise       = a $/ b
```

Code 4.10: Source Code for Canceling the Coefficient from the Highest Order

In Code 4.11, we create a new `data type` called `ODESolverFormat`. The `ODESolverFormat` contains information for the system of first-order ODEs. The `coeffVects`, `unknownVect`, and `constantVect` are responsible for $\boldsymbol{A}$, $\boldsymbol{x}$, and $\boldsymbol{c}$ in Equation 4.1.6. The `makeAODESolverFormat` is a smart constructor to create an `ODESolverFormat` by providing a `DifferentialModel`.

```haskell
-- Acceptable format for ODE solvers
-- X' = AX + c
-- coeffVects is A - coefficient matrix with identity matrix
-- unknownVect is X - unknown column vector after reduce the highest order
-- constantVect is c - constant column vector with identity matrix,
-- X' is a column vector of first-order unknowns
data ODESolverFormat = X'{
  coeffVects :: [[Expr]],
  unknownVect :: [Integer],
  constantVect :: [Expr]
}

-- | Construct an ODESolverFormat for solving the ODE.
makeAODESolverFormat :: DifferentialModel -> ODESolverFormat
makeAODESolverFormat dm = X' transEs transUnks transConsts
  where transUnks = transUnknowns $ dm ^. unknowns
        transEs = addIdentityCoeffs [transCoefficients $ head (dm ^. coefficients)] (length transUnks) 0
        transConsts = addIdentityConsts [head (dm ^. dmConstants) `divideCosntants` head (head (dm ^. coefficients))] (length transUnks)
```

Code 4.11: Source Code for Generating Equation 4.1.6

```haskell
-- Information for solving an initial value problem
data InitialValueProblem = IVP{
  initTime :: Expr, -- inital time
  finalTime :: Expr, -- end time
  initValues :: [Expr] -- initial values
}
```

Code 4.12: Source Code for IVP Infomation

In Chapter 3, we mentioned we would solve the ODE as an IVP. In Code 4.12, we create `InitialValueProblem` to store IVP-related information, including the initial time, final time and initial values.

Lastly, in Code 4.13, we create a new smart constructor to generate the `ODEInfo` automatically. In `odeInfo'`, the first parameter is `[CodeVarChunk]`. There will likely be other variables in the ODE. The `[CodeVarChunk]` contains variables other than the dependent and independent variable. The ODEOptions instructs the Drasil Code Generator on how external libraries solve the ODE. The `DifferentialModel` contains core information for the higher-order ODE. Lastly, the `InitialValueProblem` contains information for solving the ODE numerically. The `createFinalExpr` creates multiple expressions in a list. Those expressions were created based on information on the system of first-order ODEs. The `formEquations` take the following parameters: the coefficient matrix $\boldsymbol{A}$ (`[[Expr]]`), unknown vector $\boldsymbol{x}$ (combining `[Unknown]` and `ConstrConcept`), and constant vector $\boldsymbol{c}$ (`[Expr]`). Those parameters contain enough information to create explicit equations for the Drasil Code Generator. We can create explicit equations by taking the dot product of $\boldsymbol{A}$ and $\boldsymbol{x}$, and then adding the constant term. The `formEquations` will output a list of expressions equivalent to Equation 4.1.5. Once explicit equations for the higher-order ODE are created, we

can pass them to the Drasil Code Generator.

```
odeInfo' :: [CodeVarChunk] -> ODEOptions -> DifferentialModel ->
 ↪  InitialValueProblem -> ODEInfo
odeInfo' ovs opt dm ivp = ODEInfo
  (quantvar $ _indepVar dm)
  (quantvar $ _depVar dm)
  ovs
  (expr $ initTime ivp)
  (expr $ finalTime ivp)
  (map expr $ initValues ivp)
  (createFinalExpr dm)
  opt

createFinalExpr :: DifferentialModel -> []
createFinalExpr dm = map expr $ formEquations (coeffVects ode)
 ↪  (unknownVect ode) (constantVect ode) (_depVar dm)
  where ode = makeAODESolverFormat dm

formEquations :: [[Expr]] -> [Unknown] -> [Expr] ->
 ↪  ConstrConcept-> [Expr]
formEquations [] _ _ _ = []
formEquations _ [] _ _ = []
formEquations _ _ [] _ = []
formEquations (ex:exs) unks (y:ys) depVa =
  (if y == exactDbl 0 then finalExpr else finalExpr `addRe` y) :
 ↪  formEquations exs unks ys depVa
  where indexUnks = map (idx (sy depVa) . int) unks -- create X
        filteredExprs = filter (\x -> fst x /= exactDbl 0) (zip ex
 ↪  indexUnks) -- remove zero coefficients
        termExprs = map (uncurry mulRe) filteredExprs -- multiple
 ↪  coefficient with depend variables
        finalExpr = foldl1 addRe termExprs -- add terms together
```

Code 4.13: Source Code for Generating ODEInfo

Chapter 5

Summary of Future Work

5.1 Automate the Generating Process for A System of Higher-order ODEs

Chapter 4 discusses how to automate generating process for a single higher-order linear ODE. A system of higher-order ODEs has multiple higher-order ODEs in one system. Generally, we can transform each individual higher-order ODE into a system of first-order ODE. This result in multiple system of first-order ODEs. We have to combine them into one system. The difficulty may come from how to represent a nonlinear ODE. If we have a system of higher-order linear ODE, we can do steps in Chapter 4 to transform a higher-order linear ODE into a system of first-order ODE. However, more research is still needed to tackle a nonlinear ODE.

5.2 Generate an ODE Solver Object that Returns Solution on Demand

Drasil has options for generating a runnable program or standalone libraries. However, all the current ODE-related case studies generate a runnable program. The requirements specify the output to be calculated. Chapter 3 analyzes each selected external library and provides two additional specifications. One of them outputs the ODE as a function that can return the value of the dependent variable for any value of the independent variable. If other outside programs want to utilize generated code, this requirement can be the starting point.

5.3 Display ODEs in Various Forms

In Figure 2.2, we display the ODE in two different forms. In future, we want to display ODE in other forms. For example, in mathematical expression, sometimes we move all the terms to the left-hand side and leave the right-hand side to 0, as the Figure 5.1. This is just one example of the variability of displaying ODEs. More

Equation	$\frac{d^2 y_\mathrm{t}}{dt^2} + (1 + K_\mathrm{d})\frac{dy_\mathrm{t}}{dt} + (20 + K_\mathrm{p})y_\mathrm{t} + r_\mathrm{t} K_\mathrm{p} = 0$

Figure 5.1: Options of Displaying an ODE

research is still needed to allow Drasil to display ODEs in various forms.

5.4 Allow Adaptive Steps

While we were solving the ODE as an IVP numerically, we set the step size to a fixed step size. In reality, the step size can be adaptive rather than fixed. We can solve with an adaptive solver and still output the results at a fixed step size. We found some algorithms use a fixed step size for calculating numerical solutions, and others use an adaptive step size. We add the step size with the current time value to calculate the next value of dependent variables. A fixed step size means the step size is the same in each iteration. An adaptive step size means the step size is not always the same and could change based on other factors. In Table A.3, the ACM library divides algorithms into one group that uses a fixed step and others that uses an adaptive step. This discovery can further influence the design choice of solving ODE numerically in the Drasil framework. Currently, Drasil treats all step sizes as a fixed value, and it would be ideal to allow the step size to be either fixed or adaptive in the future.

5.5 Solve ODEs as a BVP

Currently, we solve ODEs as an IVP in Drasil. Solving the ODE as a BVP is another option. The starting point could be to find a suitable external library that can solve the ODE as a BVP. Then, we can generate code to link with this library. Lastly, the input of initial conditions was designed for IVPs, so a new design is required to accept both initial values and boundary values.

5.6 Handle Dependency

Once the Drasil framework generates code, the generated code relies on external libraries to calculate an ODE. In the current setting, the Drasil framework keeps copies of external libraries in the repository. In the long run, this is not practical because of the amount of space external libraries occupy. Moreover, external libraries are not currently shared across case studies, and each case study will have its own copy of external libraries. The current research has uncovered that the current way of handling dependencies in the Drasil framework is problematic. In the future, the team would like to find a better way to handle dependencies. We used a temporary solution, symbolic links, to share external libraries without duplications. By creating a symbolic link file, external libraries become sharable. In the future, the team will conduct further studies to tackle this problem more elegantly.

5.7 Define ODE Solver in Drasil

Our ultimate goal is to write the ODE solver in the Drasil language. Currently, the ODE solver is four selected external libraries. They help us to produce a numerical solution. In future, we want to remove all external libraries and design Drasil's ODE solvers to solve ODEs. Capturing the full knowledge of ODE solvers will take considerable time, but the generated code may be able to take advantage of problem-specified parameters to achieve efficiency not possible with general purpose code.

Chapter 6

Conclusion

An ODE is a type, and it exists in many forms. Previously to this research, the Drasil Framework had no flexible and reusable structure for capturing ODE information. The Drasil teams have to manually extract useful information from the original ODE to instruct the Drasil Code Generator to generate code. This approach propagates duplicated information and loses traceability. The newly created structure, `DifferentialModel`, stores linear ODE information based on the conventional matrix concept. It provides the flexibility to transform a linear ODE from one form to another mathematically equivalent form. Once we capture the knowledge of ODE in the new structure, we can reuse it for other purposes, such as producing the numerical solution and displaying the ODE.

Along with `DifferentialModel`, four selected external libraries are responsible for producing the numerical solution for a system of first-order ODEs. Drasil users can get a numerical solution by choosing an algorithm. Currently, although the Calculations module outputs a finite stream of real numbers, $\mathbb{R}^m$, there are other design options. The C# OSLO provides an option to output an infinite stream of real numbers,

$\mathbb{R}^\infty$. It has richer data than $\mathbb{R}^m$. Also, outputting the ODE as a function that can return the value of the dependent variable for any value of the independent variable could help generate libraries in Drasil. We did not complete implementing the new specifications for this, but the analysis provides a starting point for future research.

`DifferentialModel` provides reusable ODE information, and external libraries provide mathematical knowledge for solving the ODE. Before we bridge the gap between `DifferentialModel` and external libraries, we enable solving any higher-order ODEs with manually written equations via external libraries. The Double Pendulum case study demonstrates the Drasil Framework can generate code that solves a system of higher-order nonlinear ODEs. With all implementations, we are ready to bridge the gap by automating the process of extracting useful information from `DifferentialModel` and then forming an `ODEInfo`. While we are solving a single higher-order linear ODE, we generate the `ODEInfo` instead of manually creating it. The automation removes the duplicated information and potentially increases traceability.

This research accomplishes three main goals. Firstly, we capture the knowledge of linear ODE in a flexible and reusable structure. Secondly, we expand the Drasil capability to solve any higher-order ODEs with manually written equations. The last one is removing the duplicated information caused by the implementation of solving ODEs.

Appendix A

My Appendix

This appendix lists tables and code to further explain some parts of this report.

A.1 Constructors of DifferentialModel

```
-- K_d is qdDerivGain
-- y_t is opProcessVariable
-- K_p is qdPropGain
-- r_t is qdSetPointTD
imPDRC :: DifferentialModel
imPDRC = makeASingleDE
  time
  opProcessVariable
  lhs
  rhs
  "imPDRC"
  (nounPhraseSP
↪  "Computation of the Process Variable as a function of time")
  EmptyS
  where
  lhs = [exactDbl 1 `addRe` sy qdDerivGain $* (opProcessVariable
↪  $^^ 1)]
  $+ (exactDbl 1 $* (opProcessVariable $^^ 2))
  $+ (exactDbl 20 `addRe` sy qdPropGain $* (opProcessVariable $^^
↪  0))
  rhs = sy qdSetPointTD `mulRe` sy qdPropGain
```

Code A.1: Using Input Language for the Example 2.2.2 in DifferentialModel

A.2 Numerical Solution Implementation

```
def func_y_t(K_d, K_p, r_t, t_sim, t_step):
    def f(t, y_t):
        return [y_t[1], -(1.0 + K_d) * y_t[1] + -(20.0 + K_p) *
        ↪  y_t[0] + r_t * K_p]

    r = scipy.integrate.ode(f)
    r.set_integrator("dopri5", atol=Constants.Constants.AbsTol,
    ↪  rtol=Constants.Constants.RelTol)
    r.set_initial_value([0.0, 0.0], 0.0)
    y_t = [[0.0, 0.0][0]]
    while r.successful() and r.t < t_sim:
        r.integrate(r.t + t_step)
        y_t.append(r.y[0])

    return y_t
```

Code A.2: Source Code of Solving PDController in Scipy

```
1  public static ArrayList<Double> func_y_t(double K_d, double K_p,
   ↪  double r_t, double t_sim, double t_step) {
2    ArrayList<Double> y_t;
3    ODEStepHandler stepHandler = new ODEStepHandler();
4    ODE ode = new ODE(K_p, K_d, r_t);
5    double[] curr_vals = {0.0, 0.0};
6
7    FirstOrderIntegrator it = new DormandPrince54Integrator(t_step,
   ↪  t_step, Constants.AbsTol, Constants.RelTol);
8    it.addStepHandler(stepHandler);
9    it.integrate(ode, 0.0, curr_vals, t_sim, curr_vals);
10   y_t = stepHandler.y_t;
11
12   return y_t;
13 }
```

Code A.3: A Linear System of First-order Representation in ACM

```
vector<double> func_y_t(double K_d, double K_p, double r_t, double t_sim, double t_step) {
  vector<double> y_t;
  ODE ode = ODE(K_p, K_d, r_t);
  vector<double> currVals{0.0, 0.0};
  Populate pop = Populate(y_t);

  boost::numeric::odeint::runge_kutta_dopri5<vector<double>> rk = boost::numeric::odeint::runge_kutta_dopri5<vector<double>>();
  auto stepper = boost::numeric::odeint::make_controlled(Constants::AbsTol, Constants::RelTol, rk);
  boost::numeric::odeint::integrate_const(stepper, ode, currVals, 0.0, t_sim, t_step, pop);

  return y_t;
}
```

Code A.4: A Linear System of First-order Representation in ODEINT

A.3 Algorithm in External Libraries

Name	Description
`zvode`	Complex-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
`lsoda`	Real-valued Variable-coefficient Ordinary Differential Equation solver, with fixed-leading-coefficient implementation. It provides automatic method switching between implicit Adams method (for non-stiff problems) and a method based on backward differentiation formulas (BDF) (for stiff problems).
`dopri5`	This is an explicit runge-kutta method of order (4)5 due to Dormand & Prince (with stepsize control and dense output).
`dop853`	This is an explicit runge-kutta method of order 8(5,3) due to Dormand & Prince (with stepsize control and dense output).

Table A.1: Algorithm Options in Scipy - Python (15)

Name	Description
`RK547M`	This method is most appropriate for solving non-stiff ODE systems. It is based on classical Runge-Kutta formulae with modifications for automatic error and step size control.
`GearBDF`	It is an implementation of the Gear back differentiation method, a multi-step implicit method for stiff ODE systems solving.

Table A.2: Algorithm Options in OSLO - C# (13)

Step Size	Name	Description
Fixed Step	`Euler`	This class implements a simple Euler integrator for Ordinary Differential Equations.
	`Midpoint`	This class implements a second order Runge-Kutta integrator for Ordinary Differential Equations.
	`Classical RungeKutta`	This class implements the classical fourth order Runge-Kutta integrator for Ordinary Differential Equations (it is the most often used Runge-Kutta method).
	`Gill`	This class implements the Gill fourth order Runge-Kutta integrator for Ordinary Differential Equations.
	`Luther`	This class implements the Luther sixth order Runge-Kutta integrator for Ordinary Differential Equations.
Adaptive Stepsize	`Higham and Hall`	This class implements the 5(4) Higham and Hall integrator for Ordinary Differential Equations.
	`DormandPrince 5(4)`	This class implements the 5(4) Dormand-Prince integrator for Ordinary Differential Equations.
	`DormandPrince 8(5,3)`	This class implements the 8(5,3) Dormand-Prince integrator for Ordinary Differential Equations.
	`Gragg-Bulirsch-Stoer`	This class implements a Gragg-Bulirsch-Stoer integrator for Ordinary Differential Equations.
	`Adams-Bashforth`	This class implements explicit Adams-Bashforth integrators for Ordinary Differential Equations.
	`Adams-Moulton`	This class implements implicit Adams-Moulton integrators for Ordinary Differential Equations.

Table A.3: Algorithm Options in Apache Commons Maths - Java (8)

Name	Description
`euler`	Explicit Euler: Very simple, only for demonstrating purpose
`runge_kutta4`	Runge-Kutta 4: The classical Runge Kutta scheme, good general scheme without error control.
`runge_kutta_cash_karp54`	Cash-Karp: Good general scheme with error estimation.
`runge_kutta_dopri5`	Dormand-Prince 5: Standard method with error control and dense output.
`runge_kutta_fehlberg78`	Fehlberg 78: Good high order method with error estimation.
`adams_bashforth_moulton`	Adams-Bashforth-Moulton: Multi-step method with high performance.
`controlled_runge_kutta`	Controlled Error Stepper: Error control for the Runge-Kutta steppers.
`dense_output_runge_kutta`	Dense Output Stepper: Dense output for the Runge-Kutta steppers.
`bulirsch_stoer`	Bulirsch-Stoer: Stepper with step size, order control and dense output. Very good if high precision is required..
`implicit_euler`	Implicit Euler: Basic implicit routine.
`rosenbrock4`	Rosenbrock 4: Solver for stiff systems with error control and dense output.
`symplectic_euler`	Symplectic Euler: Basic symplectic solver for separable Hamiltonian system.
`symplectic_rkn_sb3a_mclachlan`	Symplectic RKN McLachlan: Symplectic solver for separable Hamiltonian system with order 6.

Table A.4: Algorithm Options in ODEINT - C++ (11)

A.4 Generated Code for Double Pendulum

We altered the source code to make it more readable. In Code A.5 and Code A.6, lines 2-12 are in one line. In Code A.7 and Code A.8, lines 4-5 are in one line. Lines 9-10 are in one line.

```
1  def f(t, theta):
2    return [
3      theta[1],
4
5      (-9.8 * (2.0 * m_1 + m_2) * math.sin(theta[0]) - m_2 * 9.8 *
       ↪  math.sin(theta[0] - 2.0 * theta[2]) - 2.0 *
       ↪  math.sin(theta[0] - theta[2]) * m_2 * (theta[3] ** 2.0 *
       ↪  L_2 + theta[1] ** 2.0 * L_1 * math.cos(theta[0] -
       ↪  theta[2])))
6        / (L_1 * (2.0 * m_1 + m_2 - m_2 * math.cos(2.0 * theta[0] -
         ↪  2.0 * theta[2]))),
7
8      theta[3],
9
10     2.0 * math.sin(theta[0] - theta[2]) * (theta[1] ** 2.0 * L_1 *
       ↪  (m_1 + m_2) + 9.8 * (m_1 + m_2) * math.cos(theta[0]) +
       ↪  theta[3] ** 2.0 * L_2 * m_2 * math.cos(theta[0] -
       ↪  theta[2]))
11       / (L_2 * (2.0 * m_1 + m_2 - m_2 * math.cos(2.0 * theta[0] -
         ↪  2.0 * theta[2])))
12     ]
```

Code A.5: Generate Python Code for Double Pendulum

```
Func<double, Vector, Vector> f = (t, theta_vec) => {
  return new Vector(
    theta_vec[1],

    (-9.8 * (2.0 * m_1 + m_2) * Math.Sin(theta_vec[0]) - m_2 * 9.8
    * Math.Sin(theta_vec[0] - 2.0 * theta_vec[2]) - 2.0 *
    Math.Sin(theta_vec[0] - theta_vec[2]) * m_2 *
    (Math.Pow(theta_vec[3], 2.0) * L_2 + Math.Pow(theta_vec[1],
    2.0) * L_1 * Math.Cos(theta_vec[0] - theta_vec[2])))
      / (L_1 * (2.0 * m_1 + m_2 - m_2 * Math.Cos(2.0 *
    theta_vec[0] - 2.0 * theta_vec[2]))),

    theta_vec[3],

    2.0 * Math.Sin(theta_vec[0] - theta_vec[2]) *
    (Math.Pow(theta_vec[1], 2.0) * L_1 * (m_1 + m_2) + 9.8 * (m_1
    + m_2) * Math.Cos(theta_vec[0]) + Math.Pow(theta_vec[3], 2.0)
    * L_2 * m_2 * Math.Cos(theta_vec[0] - theta_vec[2]))
      / (L_2 * (2.0 * m_1 + m_2 - m_2 * Math.Cos(2.0 *
    theta_vec[0] - 2.0 * theta_vec[2]))));
};
```

Code A.6: Generate C# Code for Double Pendulum

```
1  public void computeDeriv(double t, double[] theta, double[]
   ↪  dtheta) {
2    dtheta[0] = theta[1];
3
4    dtheta[1] = (-9.8 * (2.0 * m_1 + m_2) * Math.sin(theta[0]) - m_2
   ↪  * 9.8 * Math.sin(theta[0] - 2.0 * theta[2]) - 2.0 *
   ↪  Math.sin(theta[0] - theta[2]) * m_2 * (Math.pow(theta[3], 2.0)
   ↪  * L_2 + Math.pow(theta[1], 2.0) * L_1 * Math.cos(theta[0] -
   ↪  theta[2])))
5      / (L_1 * (2.0 * m_1 + m_2 - m_2 * Math.cos(2.0 * theta[0] -
   ↪  2.0 * theta[2])));
6
7    dtheta[2] = theta[3];
8
9    dtheta[3] = 2.0 * Math.sin(theta[0] - theta[2]) *
   ↪  (Math.pow(theta[1], 2.0) * L_1 * (m_1 + m_2) + 9.8 * (m_1 +
   ↪  m_2) * Math.cos(theta[0]) + Math.pow(theta[3], 2.0) * L_2 *
   ↪  m_2 * Math.cos(theta[0] - theta[2]))
10     / (L_2 * (2.0 * m_1 + m_2 - m_2 * Math.cos(2.0 * theta[0] -
   ↪  2.0 * theta[2])));
11 }
```

Code A.7: Generate Java Code for Double Pendulum

```
1  void ODE::operator()(vector<double> theta, vector<double> &dtheta,
   ↪  double t) {
2    dtheta.at(0) = theta.at(1);
3
4    dtheta.at(1) = (-9.8 * (2.0 * m_1 + m_2) * sin(theta.at(0)) -
   ↪  m_2 * 9.8 * sin(theta.at(0) - 2.0 * theta.at(2)) - 2.0 *
   ↪  sin(theta.at(0) - theta.at(2)) * m_2 * (pow(theta.at(3), 2.0)
   ↪  * L_2 + pow(theta.at(1), 2.0) * L_1 * cos(theta.at(0) -
   ↪  theta.at(2))))
5      / (L_1 * (2.0 * m_1 + m_2 - m_2 * cos(2.0 * theta.at(0) - 2.0
   ↪  * theta.at(2))));
6
7    dtheta.at(2) = theta.at(3);
8
9    dtheta.at(3) = 2.0 * sin(theta.at(0) - theta.at(2)) *
   ↪  (pow(theta.at(1), 2.0) * L_1 * (m_1 + m_2) + 9.8 * (m_1 + m_2)
   ↪  * cos(theta.at(0)) + pow(theta.at(3), 2.0) * L_2 * m_2 *
   ↪  cos(theta.at(0) - theta.at(2)))
10     / (L_2 * (2.0 * m_1 + m_2 - m_2 * cos(2.0 * theta.at(0) - 2.0
   ↪  * theta.at(2))));
11 }
```

Code A.8: Generate C++ Code for Double Pendulum

Bibliography

[1] Dawkins, P. Paul's Online Notes linear differential equations. https://tutorial.math.lamar.edu/Classes/DE/Definitions.aspx. Accessed: 2022-09-12.

[2] Logg, A., Mardal, K.-A., and Wells, G. N., Eds. *Automated Solution of Differential Equations by the Finite Element Method*, vol. 84 of *Lecture Notes in Computational Science and Engineering*. Springer, 2012.

[3] MacLachlan, B. A design language for scientific computing software in drasil. Master's thesis, McMaster University, Hamilton, ON, Canada, June 2020.

[4] Parnas, D. On the design and development of program families. *IEEE Transaction on Software Engineering 5*, 2 (March 1976), 1–9.

[5] Smith, S., and Chen, C. Commonality and requirements analysis for mesh generating software. *McMaster University, Hamilton, Ontario, Canada* (2004), 2.

[6] Szymczak, D., Smith, W. S., and Carette, J. Position paper: A knowledge-based approach to scientific software development. In *Proceedings*

of SE4Science'16 in conjunction with the International Conference on Software Engineering (ICSE) (Austin, Texas, United States, May 2016). 4 pp.

[7] The Apache Software Foundation. Commons math: The Apache commons mathematics library. https://commons.apache.org/proper/commons-math. Accessed: 2022-09-12.

[8] The Apache Software Foundation. Commons math: The Apache ordinary differential equations integration. https://commons.apache.org/proper/commons-math/userguide/ode.html. Accessed: 2022-09-12.

[9] The Drasil Research Team. Drasil. https://jacquescarette.github.io/Drasil/. Accessed: 2022-09-12.

[10] The ODEINT Project. ODEINT solving odes in c++. http://headmyshoulder.github.io/odeint-v2/. Accessed: 2022-09-12.

[11] The ODEINT Project. ODEINT v2 solving ordinary differential equations in c++. https://www.codeproject.com/Articles/268589/odeint-v2-Solving-ordinary-differential-equations. Accessed: 2022-09-12.

[12] The OSLO Project. OSLO open solving library for odes. https://www.microsoft.com/en-us/research/project/open-solving-library-for-odes/. Accessed: 2022-09-12.

[13] The OSLO Project. OSLO open solving library for odes (oslo) 1.0 user guide, getting started, step 7. https://www.microsoft.com/en-us/research/wp-content/uploads/2014/07/osloUserGuide.pdf. Accessed: 2022-09-12.

[14] The SciPy Project. SciPy.org. https://www.scipy.org/. Accessed: 2022-09-12.

[15] The SciPy Project. SciPy.org scipy.integrate.ode. https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html. Accessed: 2022-09-12.

[16] Whaley, R. C., Petitet, A., and Dongarra, J. J. Automated empirical optimization of software and the ATLAS project. https://netlib.org/lapack/lawnspdf/lawn147.pdf, 2000. Accessed: 2022-09-25.

[17] Young, T., and Mohlenkamp, M. Introduction to Numerical Methods and Matlab Programming for Engineers reduction of higher order equations to systems. http://www.ohiouniversityfaculty.com/youngt/IntNumMeth/lecture29.pdf. Accessed: 2022-09-12.